目錄

apache-shiro-1.2.x-reference

Chinese translation of Apache Shiro 1.2.x Reference Manual and the other article collection. The laset version of Apache Shiro is 1.2.x. You can also see the demos of the reference at https://github.com/waylau/apache-shiro-1.2.x-reference-demos.There is also a GitBook version of the book: http://waylau.gitbooks.io/apache-shiro-1-2-x-reference or https://www.waylau.com/apache-shiro-1.2.x-reference/. Let's READ!

《Apache Shiro 1.2.x 参考手册》 中文翻译（包含了官方文档以及其他文章）。截止现在（2016-9-18）Shiro的最新版本为 1.3.2，利用业余时间对此进行翻译，并在原文的基础上，插入配图，图文并茂方便用户理解。如有勘误欢迎指正。

Get Started 如何开始阅读

选择下面入口之一：

- https://github.com/waylau/apache-shiro-1.2.x-reference 的 SUMMARY.md（源码）
- http://waylau.gitbooks.io/apache-shiro-1-2-x-reference 点击 Read 按钮（同步更新，国内访问速度一般）
- https://waylau.com/apache-shiro-1.2.x-reference/（国内访问速度快，定期更新。最后更新于 2016-2-16）

Code 源码

书中所有示例源码，移步至 https://github.com/waylau/apache-shiro-1.2.x-reference-demos

Issue 意见、建议

如有勘误、意见或建议欢迎拍砖 https://github.com/waylau/apache-shiro-1.2.x-reference/issues

Contact 联系作者:

- Blog: waylau.com
- Gmail: waylau521(at)gmail.com
- Weibo: waylau521

- Twitter: waylau521
- Github : waylau

1. Introduction 介绍

What is Apache Shiro?

Apache Shiro是一个功能强大、灵活的，开源的安全框架。它可以干净利落地处理身份验证、授权、企业会话管理和加密。

Apache Shiro的首要目标是易于使用和理解。安全通常很复杂，甚至让人感到很痛苦，但是Shiro却不是这样子的。一个好的安全框架应该屏蔽复杂性，向外暴露简单、直观的API，来简化开发人员实现应用程序安全所花费的时间和精力。

Shiro能做什么呢？

- 验证用户身份
- 用户访问权限控制，比如：
  - 判断用户是否分配了一定的安全角色。
  - 判断用户是否被授予完成某个操作的权限
- 在非 web 或 EJB 容器的环境下可以任意使用Session API
- 可以响应认证、访问控制，或者 Session 生命周期中发生的事件
- 可将一个或以上用户安全数据源数据组合成一个复合的用户 "view"(视图)
- 支持单点登录(SSO)功能
- 支持提供“Remember Me”服务，获取用户关联信息而无需登录

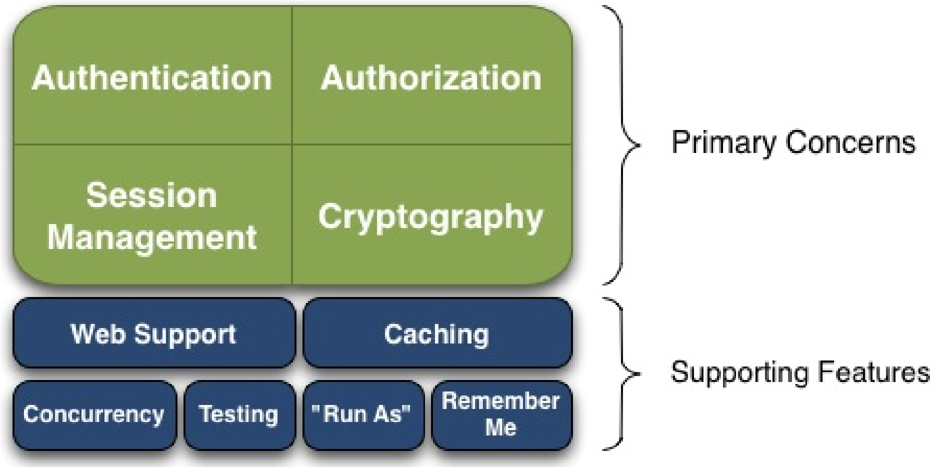
…

等等——都集成到一个有凝聚力的易于使用的API。

Shiro 致力在所有应用环境下实现上述功能，小到命令行应用程序，大到企业应用中，而且不需要借助第三方框架、容器、应用服务器等。当然 Shiro 的目的是尽量的融入到这样的应用环境中去，但也可以在它们之外的任何环境下开箱即用。

Apache Shiro Features 特性

Apache Shiro是一个全面的、蕴含丰富功能的安全框架。下图为描述Shiro功能的框架图：



Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）被 Shiro 框架的开发团队称之为应用安全的四大基石。那么就让我们来看看它们吧：

- **Authentication**（认证）：用户身份识别，通常被称为用户“登录”
- **Authorization**（授权）：访问控制。比如某个用户是否具有某个操作的使用权限。
- **Session Management**（会话管理）：特定于用户的会话管理,甚至在非web 或 EJB 应用程序。
- **Cryptography**（加密）：在对数据源使用加密算法加密的同时，保证易于使用。

还有其他的功能来支持和加强这些不同应用环境下安全领域的关注点。特别是对以下的功能支持：

- Web支持：Shiro 提供的 web 支持 api ，可以很轻松的保护 web 应用程序的安全。
- 缓存：缓存是 Apache Shiro 保证安全操作快速、高效的重要手段。
- 并发：Apache Shiro 支持多线程应用程序的并发特性。
- 测试：支持单元测试和集成测试，确保代码和预想的一样安全。
- "Run As"：这个功能允许用户假设另一个用户的身份(在许可的前提下)。
- "Remember Me"：跨 session 记录用户的身份，只有在强制需要时才需要登录。

2. Tutorial 教程

Your First Apache Shiro Application 第一个 Shiro 应用

如果你是 Apache Shiro 新手,这个简短的教程将向您展示如何通过Apache Shiro 设置一个初始和非常简单的安全应用程序。接下来我们将讨论 Shiro 的核心概念,帮助你熟悉 Shiro 的设计和 API。

当你跟随本教程时，如果你不想编辑文件,您可以获得一个几乎相同的示例作为参考。 选择一个地址:

- 在Apache Shiro 的 Subversion 存储库: https://svn.apache.org/repos/asf/shiro/trunk/samples/quickstart/
- 在Apache Shiro 的源码发布 samples/quickstart 目录中。 源码可以下载。
- （译者注：译者也提供了自己的代码，包含了中文注解，本章所含示例如下）
    - 示例1
    - 示例2
    - 示例3

Setup 设置

在这个简单的示例中,我们将创建一个非常简单的命令行应用程序,它将运行并迅速退出,这样你可以领略到 Shiro 的API。

任何应用程序

*Apache Shiro*设计从一开始就支持任何应用程序——从最小的命令行应用程序最大的集群 *web* 应用程序。对于本教程，尽管我们创建一个简单的应用程序,你都知道运用相同的使用模式来进行应用程序创建或部署。

本教程需要 Java 1.5 或更高版本。 我们还将使用 Apache Maven 作为我们的构建工具,当然这对于 Apache Shiro 来说不是必须使用。你可能获得 Shiro 的 jars,把他们以任何方式你喜欢到您的应用程序,例如也许使用Apache Ant 和 Ivy 。

对于本教程,请确保您使用 Maven 2.2.1 或更高版本。为了测试 Maven 安装是否正确，命令行下运行 mvn --version 并看到类似如下:

```
C:\Users\Administrator>mvn --version
Apache Maven 3.2.1 (ea8b2b07643dbb1b84b6d16e1f08391b666bc1e9; 2014-02-15T01:37:5
2+08:00)
Maven home: D:\Program Files (x86)\Apache Software Foundation\apache-maven-3.2.1

Java version: 1.7.0_06, vendor: Oracle Corporation
Java home: C:\Program Files (x86)\Java\jdk1.7.0_06\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 7", version: "6.1", arch: "x86", family: "windows"
```

现在,在你的文件系统中创建一个新目录,例如, shiro-tutorial 作为项目名并在目录下保存以下 Maven `pom.xml` 文件:

```
<?xml version="1.0" encoding="UTF-8"?>
    <project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org
/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>org.apache.shiro.tutorials</groupId>
    <artifactId>shiro-tutorial</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <name>First Apache Shiro Application</name>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.0.2</version>
                <configuration>
                    <source>1.5</source>
                    <target>1.5</target>
                    <encoding>${project.build.sourceEncoding}</encoding>
                </configuration>
            </plugin>

        <!-- This plugin is only to test run our little application.  It is not
             needed in most Shiro-enabled applications: -->
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.1</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>java</goal>
```

```
                    </goals>
                </execution>
            </executions>
            <configuration>
                <classpathScope>test</classpathScope>
                <mainClass>Tutorial</mainClass>
            </configuration>
        </plugin>
    </plugins>
</build>

<dependencies>
    <dependency>
        <groupId>org.apache.shiro</groupId>
        <artifactId>shiro-core</artifactId>
        <version>1.1.0</version>
    </dependency>
    <!-- Shiro uses SLF4J for logging.  We'll use the 'simple' binding
         in this example app.  See http://www.slf4j.org for more info. -->
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
        <version>1.6.1</version>
        <scope>test</scope>
    </dependency>
</dependencies>

</project>
```

教程中的 class

我们将运行一个简单的命令行应用程序,因此我们将需要创建一个带 `public static void main(String[] args)` 方法 Java 类。

包含 pom.xml 文件的同一个目录下,创建一个*src/main/java 子目录。 在 src/main/java 创建一个 Tutorial.java 文件，包含以下内容:

src/main/java/Tutorial.java

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.*;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.session.Session;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Tutorial {

    private static final transient Logger log = LoggerFactory.getLogger(Tutorial.class
);

    public static void main(String[] args) {
        log.info("My First Apache Shiro Application");
        System.exit(0);
    }
}
```

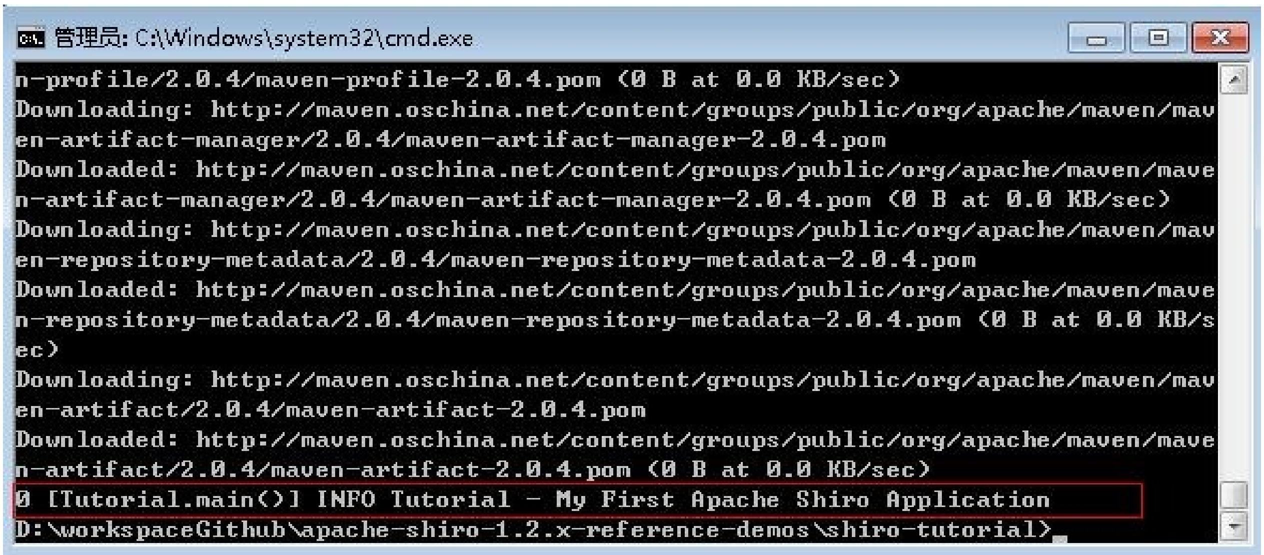
先不要管引入包的问题。下午将会很快提到。我们先测试下这个应用，会输出 "My First Apache Shiro Application" 并且退出。

Test Run 测试运行

在教程项目的根目录(如 shiro-tutorial)执行以下命令提示符中,输入以下:

> mvn compile exec:java

你就会看到我们的小教程应用程序的运行和退出。 您应当会看到类似于下面的输出（译者注：红框中的内容）



我们已经验证了应用程序成功运行——现在让我们使 Apache Shiro。当我们继续学习教程,每次我们添加更多的代码之后,您可以运行 mvn compile exec:java 看到我们的变化的结果。

Enable Shiro 使用

使用 Shiro 要理解的第一件事情是 Shiro 几乎所有的事情都和一个中心组件 SecurityManager 有关，对于那些熟悉 Java security 的人请注意：这和 java.lang.SecurityManager 不是一回事。

我们将在Architecture章节详细描述 Shiro 的设计，但现在有必要知道 Shrio SecurityManager 是程序中 Shiro 的核心，每一个程序都必定会存在一个 SecurityManager，所以，在我们这个示例程序中必须做的第一件事情是建立一个 SecurityManager 实例。

Configuration 配置

虽然我们可以直接对 SecurityManager 实例化，但在 Java 代码中对Shiro 的 SecurityManager 所须的选项和内部组件进行配置会让人感觉有点小痛苦--而将这些 SecurityManager 配置用一个灵活的配置文件实现就会简单地多。

为此，Shiro 默认提供了一个基本的 INI 配置文件的解决方案，人们已经对庞大的 XML 文件有些厌倦了，而一个 INI 文件易读易用，而且所依赖的组件很少，稍后你就会通过一个简单易懂的示例明白 INI 在对简单对象进行配置的时候是非常有效率的，比如 SecurityManager

多种配置选择

Shiro 的 *SecurityManager* 的实现和其所依赖的组件都是 *JavaBean*，所以可以用多种形式对 *Shiro* 进行配置，比如*XML*（*Spring, JBoss, Guice,* 等等），*YAML*, *JSON, Groovy Builder markup*，及其它，*INI* 只是 *Shiro* 一种最基本的配置方式，使得其可以在任何环境中进行配置比如在那些没有以上配置形式的环境中。

shiro.ini

在这个示例中我们使用一个 INI 文件来配置Shiro SecurityManager，首先，在 pom.xml 同目录中创建一个src/main/resources子目录，在该子目录中创建一个 shiro.ini 文件，内容如下：

src/main/resources/shiro.ini

```
# =============================================================================
# Tutorial INI configuration
#
# Usernames/passwords are based on the classic Mel Brooks' film "Spaceballs" :)
# =============================================================================

# -----------------------------------------------------------------------------
# Users and their (optional) assigned roles
# username = password, role1, role2, ..., roleN
# -----------------------------------------------------------------------------
[users]
root = secret, admin
guest = guest, guest
presidentskroob = 12345, president
darkhelmet = ludicrousspeed, darklord, schwartz
lonestarr = vespa, goodguy, schwartz

# -----------------------------------------------------------------------------
# Roles with assigned permissions
# roleName = perm1, perm2, ..., permN
# -----------------------------------------------------------------------------
[roles]
admin = *
schwartz = lightsaber:*
goodguy = winnebago:drive:eagle5
```

可以看到，在该配置文件中最基础地配置了几个静态的帐户，对我们这一个程序已经足够了，在以后的章节中，将会看到如何使用更复杂的用户数据比如数据库、LDAP 和活动目录等。

Referencing the Configuration 引用配置

现在我们已经定义了一个 INI 文件，我们可以在我们的示例程序中创建SecurityManager 实例了，将 main 函数中的代码进行如下调整：

```
public static void main(String[] args) {

    log.info("My First Apache Shiro Application");

    //1.
    Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.
ini");

    //2.
    SecurityManager securityManager = factory.getInstance();

    //3.
    SecurityUtils.setSecurityManager(securityManager);

    System.exit(0);
}
```

这就是我们要做的--仅仅使用三行代码就把Shiro加进了我们的程序，就是这么简单。

执行mvn compile exec:java 可以看到程序成功的运行（由于 Shiro 默认在 debug 或更底层才记录日志，所以你不会看到任何 Shiro 的日志输出--只要运行时没有错误提示，你就可以知道已经成功了）。

上面所加入的代码做了下面的事情：

1. 使用 Shiro 的 IniSecurityManagerFactory 加载了我们的shiro.ini 文件，该文件存在于 classpath 根目录里。这个执行动作反映出 shiro 支持 Factory Method Design Pattern（工厂模式）。classpath：资源的指示前缀，告诉 shiro 从哪里加载 ini 文件（其它前缀，如 url:和 file: 也被支持）。

2.factory.getInstance() 方法被调用，该方法分析 INI 文件并根据配置文件返回一个 SecurityManager 实例。

3.在这个简单示例中，我们将 SecurityManager 设置成了static (memory) singleton，可以通过 JVM 访问，注意如果你在一个 JVM 中加载多个使用 shiro 的程序时不要这样做，在这个简单示例中，这是可以的，但在其它成熟的应用环境中，通常会将 SecurityManager 放在程序指定的存储中（如在 web 中的 ServletContexct 或者 Spring、Guice、 JBoss DI 容器实例）中。

Using Shiro 使用

现在我们的 SecurityManager 已经准备好了，我们可以开始进行我们真正关心的事情--执行安全操作了。

为了保护我们的程序安全，我们或许问自己最多的问题就是“谁是当前的用户？”或者“当前用户是否允许做某件事？”通常我们会在写代码或者设计用户接口的时候问这些问题：程序通常建立在用户基础上，程序功能展示（和安全）也基于每一个用户。所以，通常我们考虑我们程序安全的方法也建立在当前用户的基础上，Shiro 的 API 提供了'the current user'概念，即 Subject。

在几乎所有的环境中，你可以通过如下语句得到当前用户的信息：

```
Subject currentUser = SecurityUtils.getSubject();
```

使用 SecurityUtils.getSubject()，我们可以获取当前执行的Subject，Subject是一个安全术语意思是“当前运行用户的指定安全视图（a security-specific view of the currently executing user）”，这里并不称之为“User”因为“User”这个词通常和一个人相关，但在安全认证中，“Subject”可以认为是一个人，也可以认为是第三方进程、时钟守护任务、守护进程帐户或者其它。它可简单描述为“当前和软件进行交互的事件”，在大多数情况下，你可以认为它是一个“人（User）”。

在一个独立的程序中调用 getSubject() 会在程序指定位置返回一个基于用户数据的 Subject，在服务器环境（如 web 程序）中，它将获取一个和当前线程或请求相关的基于用户数据的 Subject。

现在你得到了Subject，你可以利用它做什么呢？

如果你针对该用户希望一些事情在程序当前会话期内可行，你可以获取他们的 session：

```
Session session = currentUser.getSession();
session.setAttribute( "someKey", "aValue" );
```

Session 是 shiro 指定的一个实例，提供基本上所有 HttpSession 的功能，但具备额外的好处和不同：它不需要一个 HTTP 环境！

如果发布到一个 web 程序中，默认情况下 Session 将会使用HttpSession 作为基础，但是，在一个非 web 程序中，比如该简单示例程序中，Shiro 将自动默认使用它的 Enterprise Session Management，这意味着你可以在任何程序中使用相同的 API，而根本不需要考虑发布环境！这打开了一个全新的世界，从此任何需要 session 的程序不再需要强制使用 HttpSession 或者 EJB Stateful Session,并且，终端可以共享 session 数据。

现在你可以获取一个 Subject 和它们的 Session，真正填充有用的代码如检测其是否被允许做某些事情如何？比如检查其角色和权限？

我们只能对一个已知用户做这些检测，如上我们获取 Subject 实例表示当前用户，但是当前用户是认证，嗯，他们是任何人--直到他们至少登录一次，我们现在就做这件事情：

```
if ( !currentUser.isAuthenticated() ) {
    //收集用户的主要信息和凭据，来自GUI中的特定的方式
    //如包含用户名/密码的HTML表格，X509证书，OpenID，等。
    //我们将使用用户名/密码的例子因为它是最常见的。.
    UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");

    //支持'remember me' (无需配置，内建的!):
    token.setRememberMe(true);

    currentUser.login(token);
}
```

就是这样，不能再简单了。

但如果登录失败了呢，你可以捕获所有异常然后按你期望的方式去处理：

```
try {
    currentUser.login( token );
    //无异常，说明就是我们想要的!
} catch ( UnknownAccountException uae ) {
    //username 不存在，给个错误提示?
} catch ( IncorrectCredentialsException ice ) {
    //password 不匹配，再输入?
} catch ( LockedAccountException lae ) {
    //账号锁住了，不能登入。给个提示?
}
    ... 更多类型异常 ...
} catch ( AuthenticationException ae ) {
    //未考虑到的问题 - 错误?
}
```

这里有许多不同类别的异常你可以检测到，也可以抛出你自己异常。详见 AuthenticationException JavaDoc

小贴士：

最好的方式是将普通的失败信息反馈给用户，你总不会希望帮助黑客来攻击你的系统吧。

好，到现在为止，我们有了一个登录用户，接下来我们还可以做什么？

让我们显示他们是谁

```
//打印主要信息 (本例子是 username):
log.info( "User [" + currentUser.getPrincipal() + "] logged in successfully." );
```

我们也可以判断他们是否拥有某个角色：

```
if ( currentUser.hasRole( "schwartz" ) ) {
    log.info("May the Schwartz be with you!" );
} else {
    log.info( "Hello, mere mortal." );
}
```

我们也可以判断他们是否拥有某个特定动作或入口的权限：

```
if ( currentUser.isPermitted( "lightsaber:weild" ) ) {
    log.info("You may use a lightsaber ring.  Use it wisely.");
} else {
    log.info("Sorry, lightsaber rings are for schwartz masters only.");
}
```

同样，我们还可以执行非常强大的 instance-level （实例级别）的权限检测，检测用户是否具备访问某个类型特定实例的权限：

```
if ( currentUser.isPermitted( "winnebago:drive:eagle5" ) ) {
    log.info("You are permitted to 'drive' the 'winnebago' with license plate (id) 'ea
gle5'.  " +
                "Here are the keys - have fun!");
} else {
    log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
}
```

轻而易举，是吧！

最后，当用户不再使用系统，可以退出登录：

```
currentUser.logout(); //清楚识别信息，设置 session 失效.
```

Final Tutorial class 最终的 class

在加入上述代码后，下面的就是我们完整的文件，你可以自由编辑和运行它，可以尝试改变安全检测（以及INI配置）：

src/main/java/Tutorial.java

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.authc.*;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.session.Session;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Tutorial {

    private static final transient Logger log = LoggerFactory.getLogger(Tutorial.class
);


    public static void main(String[] args) {
        log.info("My First Apache Shiro Application");

        Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:sh
iro.ini");
        SecurityManager securityManager = factory.getInstance();
        SecurityUtils.setSecurityManager(securityManager);


        // 获取当前执行用户:
        Subject currentUser = SecurityUtils.getSubject();

        // 做点跟 Session 相关的事
        Session session = currentUser.getSession();
        session.setAttribute("someKey", "aValue");
        String value = (String) session.getAttribute("someKey");
        if (value.equals("aValue")) {
            log.info("Retrieved the correct value! [" + value + "]");
        }

        // 登录当前用户检验角色和权限
        if (!currentUser.isAuthenticated()) {
            UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vesp
a");
            token.setRememberMe(true);
            try {
                currentUser.login(token);
            } catch (UnknownAccountException uae) {
                log.info("There is no user with username of " + token.getPrincipal());
            } catch (IncorrectCredentialsException ice) {
                log.info("Password for account " + token.getPrincipal() + " was incorr
ect!");
            } catch (LockedAccountException lae) {
                log.info("The account for username " + token.getPrincipal() + " is loc
ked.  " +
                        "Please contact your administrator to unlock it.");
            }
            // ... 捕获更多异常
            catch (AuthenticationException ae) {
                //无定义?错误?
            }
        }

        //说出他们是谁:
```

```
        //打印主要识别信息 (本例是 username):
        log.info("User [" + currentUser.getPrincipal() + "] logged in successfully.");

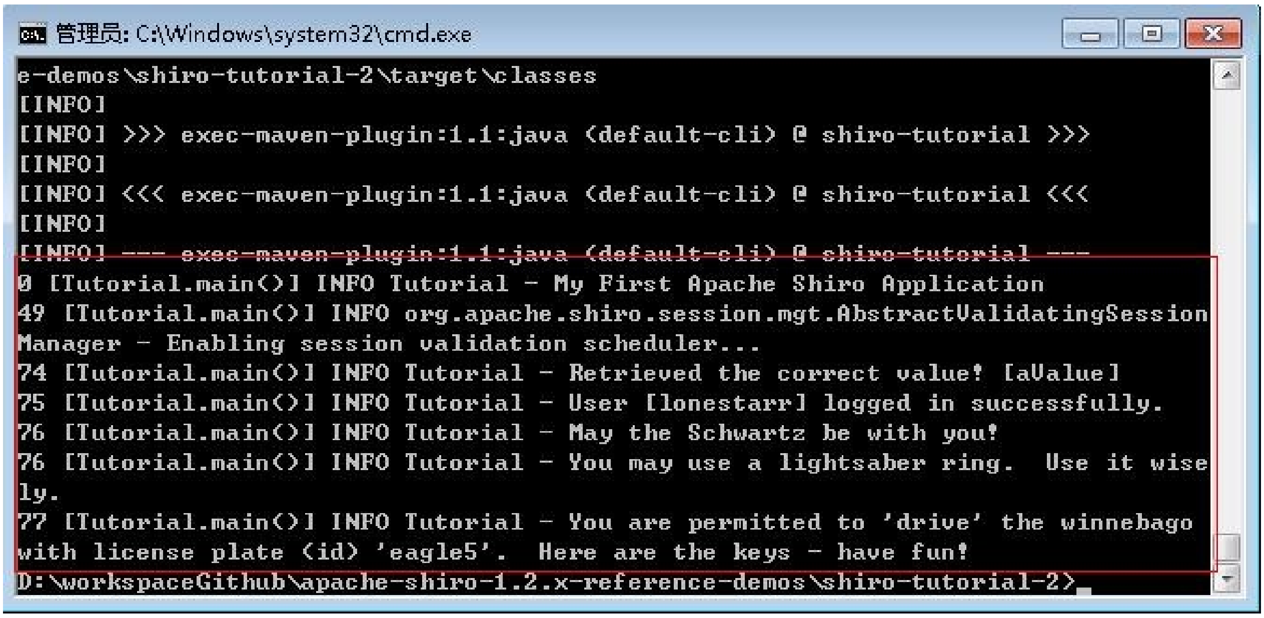
        //测试角色:
        if (currentUser.hasRole("schwartz")) {
            log.info("May the Schwartz be with you!");
        } else {
            log.info("Hello, mere mortal.");
        }

        //测试一个权限 (非（instance-level）实例级别)
        if (currentUser.isPermitted("lightsaber:weild")) {
            log.info("You may use a lightsaber ring.  Use it wisely.");
        } else {
            log.info("Sorry, lightsaber rings are for schwartz masters only.");
        }

        //一个(非常强大)的实例级别的权限:
        if (currentUser.isPermitted("winnebago:drive:eagle5")) {
            log.info("You are permitted to 'drive' the winnebago with license plate (i
d) 'eagle5'.  " +
                    "Here are the keys - have fun!");
        } else {
            log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
        }

        //完成 - 退出t!
        currentUser.logout();

        System.exit(0);
    }
}
```



总结

非常希望这示例介绍能帮助你理解如何在基础程序中加入 Shiro,并理解Shiro 的设计理念，Subject 和 SecurityManager。

但这个程序太简单了，你可能会问自己，“如果我不想使用 INI 用户帐号，而希望连接更为复杂的用户数据源呢？”

解决这些问题需要更深入地了解 并理解Shiro 的架构和配置机制，我们将在下一节 Architecture 中介绍。

3. Architecture 架构

Apache Shiro 设计理念是使程序的安全变得简单直观而易于实现，Shiro的核心设计参照大多数用户对安全的思考模式--如何对某人（或某事）在与程序交互的环境中的进行安全控制。

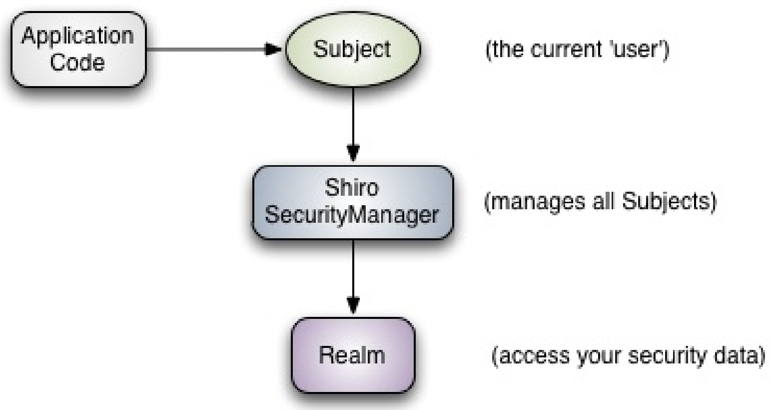
程序设计通常都以用户故事为基础，也就是说,你会经常设计用户接口或服务api基于用户如何(或应该)与软件交互。 例如,你可能会说,“如果我的应用程序的用户交互是登录,我将展示他们可以单击一个按钮来查看他们的帐户信息。 如果不登录,我将展示一个注册按钮。”

这个陈述例子指出我们开发程序很大程度上是为了满足用户的需求，即使“用户（User）”是另外一个软件系统而并非一个人，你仍然要写代码对当前与你软件交互的谁（或者什么）的动作进行回应。

Shiro 从它的设计中表现了这种理念，为了与软件开发者的直觉相配合，Apache Shiro 在几乎所有程序中保留了直观和易用的特性。

High-Level Overview 高级概述

在概念层，Shiro 架构包含三个主要的理念：Subject,SecurityManager和 Realm。下面的图展示了这些组件如何相互作用，我们将在下面依次对其进行描述。



- **Subject**:就像我们在上一章示例中提到的那样，Subject 本质上是当前运行用户特定的'View'(视图)，而单词“User”经常暗指一个人，Subject 可以是一个人，但也可以是第三方服务、守护进程帐户 、时钟守护任务或者其它--当前和软件交互的任何事件。 Subject 实例都和（也需要）一个 SecurityManager 绑定，当你和一个Subject 进行交互，这些交互动作被转换成 SecurityManager 下Subject 特定的交互动作。

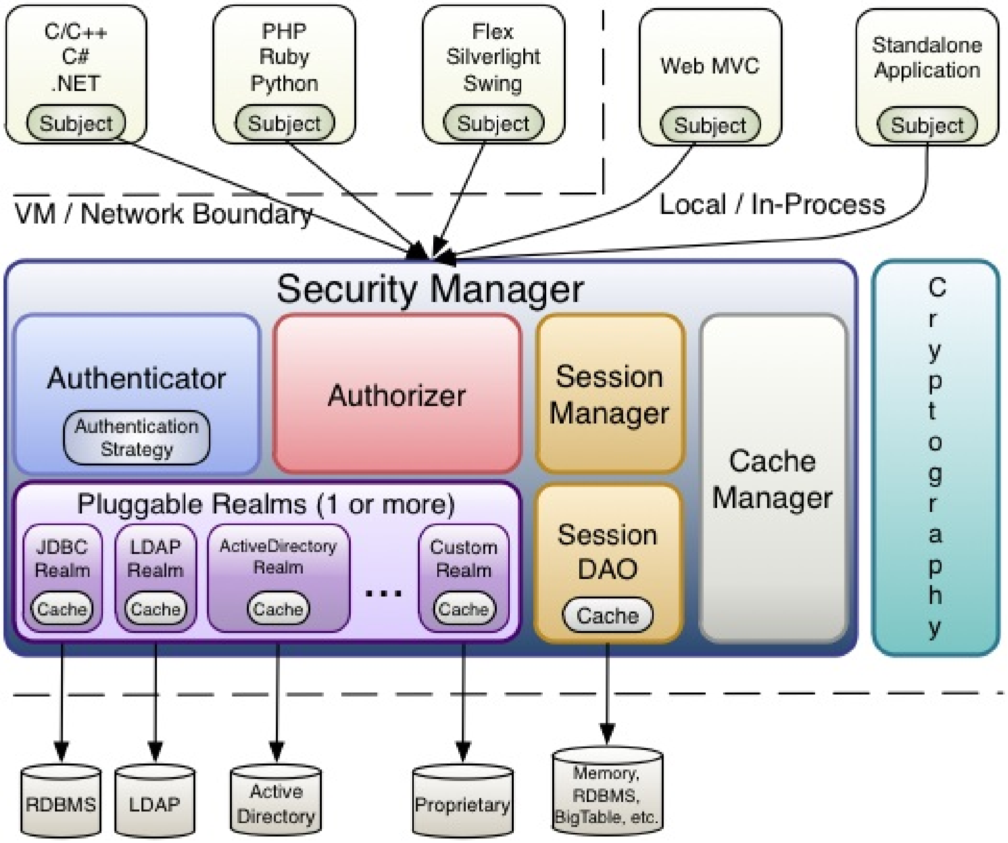
- **SecurityManager**: SecurityManager 是 Shiro 架构的核心，配合内部安全组件共同组成安全伞。然而，一旦一个程序配置好了SecurityManager 和它的内部对象，SecurityManager通常独自留下来，程序开发人员几乎花费的所有时间都集中在 Subjet

API上。我们将在以后详细讨论 SecurityManager，但当你和一个 Subject 互动时了解它是很重要的。任何 Subject 的安全操作中 SecurityManager 是幕后真正的举重者，这在上面的图表中可以反映出来。

- **Realms**： Reamls 是 Shiro 和你的程序安全数据之间的"桥"或者"连接"，它用来实际和安全相关的数据如用户执行身份认证（登录）的帐号和授权（访问控制）进行交互，Shiro 从一个或多个程序配置的 Realm 中查找这些东西。 Realm 本质上是一个特定的安全 DAO：它封装与数据源连接的细节，得到Shiro 所需的相关的数据。在配置 Shiro 的时候，你必须指定至少一个Realm 来实现认证（authentication）和/或授权（authorization）。SecurityManager 可以配置多个复杂的 Realm，但是至少有一个是需要的。 Shiro 提供开箱即用的 Realms 来连接安全数据源（或叫地址）如 LDAP、JDBC、文件配置如INI和属性文件等，如果已有的Realm不能满足你的需求你也可以开发自己的Realm实现。 和其它内部组件一样，Shiro SecurityManager 管理如何使用Realms获取 Subject 实例所代表的安全和身份信息。

Detailed Architecture 详细架构

下面的图表展示了 Shiro 的核心架构思想，下面有简单的解释。



- **Subject** (org.apache.shiro.subject.Subject)

正在与软件交互的一个特定的实体"view"（用户、第三方服务、时钟守护任务等）。

- **SecurityManager** (org.apache.shiro.mgt.SecurityManager)

如同上面提到的，SecurityManager 是 Shiro 的核心，它基本上就是一把“保护伞”用来协调它管理的组件使之平稳地一起工作，它也管理着 Shiro 中每一个程序用户的视图，所以它知道每个用户如何执行安全操作。

- **Authenticator**(org.apache.shiro.authc.Authenticator)

Authenticator 是一个组件，负责执行和反馈用户的认证（登录），如果一个用户尝试登录，Authenticator 就开始执行。Authenticator 知道如何协调一个或多个保存有相关用户/帐号信息的 Realm，从这些 Realm中获取这些数据来验证用户的身份以确保用户确实是其表述的那个人。

- **Authentication Strategy**(org.apache.shiro.authc.pam.AuthenticationStrategy)

如果配置了多个 Realm，AuthenticationStrategy 将会协调 Realm 确定在一个身份验证成功或失败的条件（例如，如果在一个方面验证成功了但其他失败了，这次尝试是成功的吗？是不是需要所有方面的验证都成功？还是只需要第一个？）

- **Authorizer**(org.apache.shiro.authz.Authorizer)

Authorizer 是负责程序中用户访问控制的组件，它是最终判断一个用户是否允许做某件事的途径，像 Authenticator 一样，Authorizer 也知道如何通过协调多种后台数据源来访问角色和权限信息，Authorizer 利用这些信息来准确判断一个用户是否可以执行给定的动作。

- **SessionManager**(org.apache.shiro.session.mgt.SessionManager)

SessionManager 知道如何创建并管理用户 Session 生命周期而在所有环境中为用户提供一个强有力的 Session 体验。这在安全框架领域是独一无二--Shiro 具备管理在任何环境下管理用户 Session 的能力，即使没有 Web/Servlet 或者 EJB 容器。默认情况下，Shiro 将使用现有的session（如Servlet Container），但如果环境中没有，比如在一个独立的程序或非 web 环境中，它将使用它自己建立的 session 提供相同的作用，sessionDAO 用来使用任何数据源使 session 持久化。

- **SessionDAO**(org.apache.shiro.session.mgt.eis.SessionDAO)

SessionDAO 代表 SessionManager 执行 Session 持久（CRUD）动作，它允许任何存储的数据挂接到 session 管理基础上。

- **CacheManager**(org.apache.shiro.cache.CacheManager)

CacheManager 为 Shiro 的其他组件提供创建缓存实例和管理缓存生命周期的功能。因为 Shiro 的认证、授权、会话管理支持多种数据源，所以访问数据源时，使用缓存来提高访问效率是上乘的选择。当下主流开源或企业级缓存框架都可以继承到 Shiro 中，来获取更快更高效的用户体验。

- **Cryptography** (org.apache.shiro.crypto.*)

Cryptography 在安全框架中是一个自然的附加产物，Shiro 的 crypto 包包含了易用且易懂的加密方式，Hashes（即digests）和不同的编码实现。该包里所有的类都亦于理解和使用，曾经用过 Java 自身的加密支持的人都知道那是一个具有挑战性的工作，而 Shiro 的加密 API 简化了 java 复杂的工作方式，将加密变得易用。

- **Realms** (org.apache.shiro.realm.Realm)

如同上面提到的，Realm 是 shiro 和你的应用程序安全数据之间的"桥"或"连接"，当实际要与安全相关的数据进行交互如用户执行身份认证（登录）和授权验证（访问控制）时，shiro 从程序配置的一个或多个Realm 中查找这些数据，你需要配置多少个 Realm 便可配置多少个 Realm（通常一个数据源一个），shiro 将会在认证和授权中协调它们。

SecurityManager

因为 Shiro API 鼓励以 Subject 为中心的开发方式，大部分开发人员将很少会和 SecurityManager 直接交互（尽管框架开发人员也许发现它非常有用），尽管如此，知道 SecurityManager 如何工作，特别是当在一个程序中进行配置的时候，是非常重要的。

Design 设计

如前所述，程序中 SecurityManager 执行操作并且管理所有程序用户的状态，在 Shiro 基础的 SecurityManager 实现中，包含以下内容：

- 认证（Authentication）
- 授权（Authorization）
- 会话管理（Session Management）
- 缓存管理（Cache Management）
- Realm协调（Realm coordination）
- 事件传导（Event propagation ）
- "RememberMe" 服务（"Remember Me" Services）
- 建立Subject(Subject creation)
- 退出登录（Logout）

及其它。

但这些功能都在一个单独的组件中管理，并且，当所有功能集中在一个类中实现是灵活和可定制是非常困难的。

为了实现配置的简单、灵活、可插拔，Shiro在设计时实现了高模块化--尽管模块化，SecurityManager（包括它的继承类）并没有做到，相反地，SecurityManager实现更像一个轻量级的'容器（container）'，代表几乎所有嵌套/封装组件的行为，这种'封装（wrapper）'设计在上面的架构图表中已有反映。

当组件执行逻辑的时候，SecurityManager 知道如何以及何时去协调组件做出正确的动作。

SecurityManager 和 JavaBean 兼容，这允许你（或者配置途径）通过标准的J avaBean 访问/设置方法（get/set）很容易地定制插件，这意味着 Shiro 模块可以根据用户行为转化成简易的配置。

简易的配置

因为适合*JavaBean*，任何支持*Javabean*配置的组件都有非常简单的途径配置*SecurityManager*，如 *Spring*、*Guice*、*JBoss,*等等。

我们将在下一节讨论配置（Configuration ）

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

4. Configuration 配置

Shiro 可以在任何环境下工作，从简单的命令行程序到大型企业级集群项目，因为环境的多样化，可以通过许多途径来配合当前环境的配置方式进行配置，在本章我们来了解一下 Shiro 核心支持的配置方式。

多种配置选择

Shiro 的 *SecurityManager* 的实现和其所依赖的组件都是 *JavaBean*，所以可以用多种形式对 *Shiro* 进行配置，比如*XML*（*Spring, JBoss, Guice,* 等等），*YAML*, *JSON, Groovy Builder markup*，及其它，*INI* 只是 *Shiro* 一种最基本的配置方式，使得其可以在任何环境中进行配置比如在那些没有以上配置形式的环境中。

Programmatic Configuration 在程序中配置

创建一个 SecurityManager 并使之可用最简单的方法就是创建一个 org.apache.shiro.mgt.DefaultSecurityManager 对象并且将它写入代码，例如：

```
Realm realm = //实例化或获得一个Realm的实例。我们将稍后讨论Realm。

SecurityManager securityManager = new DefaultSecurityManager(realm);

//使SecurityManager实例通过静态存储器对整个应用程序可见：
SecurityUtils.setSecurityManager(securityManager);
```

仅仅三行代码，你就可以拥有一个适用于任何程序的功能全面的 Shiro 环境，多么简单。

SecurityManager Object Graph

如同我们在架构（Architecture ）中讨论过的，Shiro SecurityMangger 本质上是一个由一套安全组件组成的对象模块视图（graph），因为与 JavaBean兼容，所以可以对所有这些组件调用的 getter 和 setter 方法来配置SecurityManager 和它的内部对象视图。

例如，你想用一个自定义的 SessionDAO 来定制 Session Management从而配置一个 SecurityManager 实例，你就可以使用 SessionManager 的 setSessionDAO 方法直接 set 这个 SessionDAO。

```
...

DefaultSecurityManager securityManager = new DefaultSecurityManager(realm);

SessionDAO sessionDAO = new CustomSessionDAO();

((DefaultSessionManager)securityManager.getSessionManager()).setSessionDAO(sessionDAO)
;
...
```

使用这些函数，你可以配置 SecurityManager 视图（graph）中的任何一部分。

虽然在程序中配置很简单，但它并不是我们现实中配置的完美解决方案。在几种情况下这种方法可能并不适合你的程序：

- 它需要你确切知道并实例化一个直接实现（direct implementation），然而更好的做法是你并不需要知道这些实现也不需要知道从哪里找到它们。
- 因为JAVA类型安全的特性，你必须对通过 get* 获取的对象进行强制类型转换，这么多强制转换非常的丑陋、累赘并且会和你的类紧耦合。
- SecurityUtils.setSecurityManager 方法会将 SecurityManager 实例化为虚拟机的单独静态实例，在大多数程序中没有问题，但如果有多个使用 Shiro 的程序在同一个 JVM 中运行时，各程序有自己独立的实例会更好些，而不是共同引用一块静态内存。
- 改变配置就需要重新编译你的程序。

然而，尽管有这些不足，在程序中定制的这种方法在限制内存（memory-constrained ）的环境中还是很有价值的，像智能电话程序。如果你的程序不是运行在一个限制内存的环境中，你会发现基于文本的配置会更易读易用。

INI Configuration 配置

大多数程序已经改为使用基于文本的配置，不需要依靠代码就可进行修改，对于不熟悉Shiro API的人来说，也易于理解。

为了确保具有共性的基于文本配置的途径适用于任何环境而且减少对第三方的依赖，Shiro 支持使用 INI 创建 SecurityManager 对象视图（graph）以及它支持的组件，INI 易读易配置，很容易创建并且对大多数程序都很适合。

Creating a SecurityManager from INI 通过INI资源创建 SecurityManager

这里举两个通过INI配置创建SecurityManager的例子。

SecurityManager from an INI resource 从INI资源创建SecurityManager

我们可以从一个INI资源路径创建一个 SecurityManager 实例，资源可以通过文件系统（前缀为file:）、类路径(classpath:)或者URL(url:)获得，下面的例子使用一个 Factory 从类路径根目录加载 shiro.ini 并返回一个 SecurityManager 实例。

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini"
);
SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

SecurityManager from an INI instance 通过INI实例创建SecurityManager

INI 配置可以通过org.apache.shiro.config.Ini 类用程序方式创建，这个 INI 类类似于 JDK 的 java.util.Properties类，但支持通过section 名分割。例子如下：

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.util.Factory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.config.Ini;
import org.apache.shiro.config.IniSecurityManagerFactory;

...

Ini ini = new Ini();
//populate the Ini instance as necessary
...
Factory<SecurityManager> factory = new IniSecurityManagerFactory(ini);
SecurityManager securityManager = factory.getInstance();
SecurityUtils.setSecurityManager(securityManager);
```

现在我们知道如何使用 INI 配置文件创建一个 SecurityManager，让我们仔细了解一下如何定义一个 shiro INI配置文件。

INI Sections

INI 基于文本配置，在独立命名的区域内通过成对的键名/键值组成。键名在每个区域内必须唯一，但在整个配置文件中并不需要这样（这点和JDK的Properties不同），每一个区域（section）可以看作是一个独立的Properties 定义。

注释行可以用“#”或“;”标识。

这里是一个 Shiro 可以理解的各 section 的示例。

```
# =======================
# Shiro INI configuration
# =======================

[main]
# Objects and their properties are defined here,
# Such as the securityManager, Realms and anything
# else needed to build the SecurityManager

[users]
# The 'users' section is for simple deployments
# when you only need a small number of statically-defined
# set of User accounts.

[roles]
# The 'roles' section is for simple deployments
# when you only need a small number of statically-defined
# roles.

[urls]
# The 'urls' section is used for url-based security
# in web applications.  We'll discuss this section in the
# Web documentation
```

[main]

[main]区域是配置程序 SecurityManager 实例及其支撑组件的地方，如 Realm。

通过INI配置像 SecurityManager 的对象实例及其支撑组件听起来是一件很困难的事情，因为在这里我们只能用键名/键值对。但通过定义一些对象视图（graphs）可以理解的惯例，你发现你完全可以这样做。Shiro 利用这些假定的惯例来实现一个简单而简明的配置途径。

我们经常将这种方法认为是“可怜人的（poor man's）”的依赖注入，虽然不及成熟的Spring/Guice/JBoss的XML文件强大，但你会发现它可以做很多事情而且并不复杂，当然当那配置途径也可以使用，但对 Shiro 来讲并不是必须的。

仅仅吊一下胃口，这里是一个简单的可以使用的[main]配置，下面我们会详细介绍，但你可能发现你仅凭直觉就可以理解一些。

```
[main]
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher

myRealm = com.company.security.shiro.DatabaseRealm
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
myRealm.password = secret
myRealm.credentialsMatcher = $sha256Matcher

securityManager.sessionManager.globalSessionTimeout = 1800000
```

Defining an object 定义一个对象

在[main]中包含以下片段。

```
[main]
myRealm = com.company.shiro.realm.MyRealm
...
```

这一行实例化了一个类型为 com.company.shiro.realm.MyRealm 的对象实例并且使对象使用 myRealm 作为名称以便于将来引用和配置。

如果对象实例化时实现了 org.apache.shiro.util.Nameable 接口，Nameable.setName方法将被以该名（在此例中为myRealm）命名的对象调用。

Setting object properties 设置对象属性

Primitive Values 原始值

简单的原始值属性可以使用下面的等于符号进行设置：

```
...
myRealm.connectionTimeout = 30000
myRealm.username = jsmith
...
```

这些配置行转换为方法调用就是：

```
...
myRealm.setConnectionTimeout(30000);
myRealm.setUsername("jsmith");
...
```

怎么做到的呢？它假定所有对象都是兼容 JavaBean 的 POJO。在设置这些属性时，Shiro 默认使用 Apache 通用的 BeanUtils 来完成这项复杂的工作，所以虽然 INI 值是文本，BeanUtils 知道如何将这些字符串值转换为适合的原始值类型并调用合适的 JavaBeans 的 setter 方法。

Reference Values 引用值

如果你想设置的值并不是一个原始值，而是另一个对象怎么办呢？你可以使用一个 `$` 符来引用一个之前定义的实例，如：

```
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
myRealm.credentialsMatcher = $sha256Matcher
...
```

这定义了名为 sha256Matcher 的对象并且使用 BeanUtils 将其设置到myRealm 的实例中（通过调用 myRealm.setCredentialsMatcher(sha256Matcher) 方法）。

Nested Properties 嵌套属性

通过在等号左侧使用点符号，你可以得到你希望设置对象视图最终的对象/属性，例如下面这行配置：

```
...
securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

转换逻辑为（通过BeanUtils）：

```
securityManager.getSessionManager().setGlobalSessionTimeout(1800000);
```

用这种方法访问的层数需要多深可以有多深： object.property1.property2....propertyN.value = blah

BeanUtils 属性支持

BeanUtils 支持任何指定的属性操作，在 *Shiro [main]* 区域中*setProperty*方法将被调用，包括集合（*set*）/列表（*list*）/图（*map*），查看*Apache Commons BeanUtils Website*和文档了解更多的信息。

Byte Array Values 字节数组值

因为原始的字节数组不能直接在文本中定义，我们必须使用字节数组的文本编码。可以使用64位编码（默认）或者16位编码，默认为64位编码因为使用64位编码实际文字会少一些--它拥有很大的编码表，这意味着你的标识会更短（对于文本配置来讲会好一些）。

```
# The 'cipherKey' attribute is a byte array.    By default, text values
# for all byte array properties are expected to be Base64 encoded:

securityManager.rememberMeManager.cipherKey = kPH+bIxk5D2deZiIxcaaaA==
...
```

如果你想使用16位编码，你必须在字串前面加上 `0x` 前缀：

```
securityManager.rememberMeManager.cipherKey = 0x3707344A4093822299F31D008
```

Collection Properties 集合属性

列表（Lists）、集合（Sets）、图（Maps）可以像其它属性一样设置--直接设置或者像嵌套属性一样，对于列表和集合，只需指定一个逗号分割的值集或者对象引用集。

如定义一些SessionListeners：

```
sessionListener1 = com.company.my.SessionListenerImplementation
...
sessionListener2 = com.company.my.other.SessionListenerImplementation
...
securityManager.sessionManager.sessionListeners = $sessionListener1, $sessionListener2
```

对于图（Maps），你可以指定以逗号分割的键-值对列表，每个键-值之间用冒号分割

```
object1 = com.company.some.Class
object2 = com.company.another.Class
...
anObject = some.class.with.a.Map.property

anObject.mapProperty = key1:$object1, key2:$object2
```

在上面的例子中，$object1 引用的对象将存于键 key1 之下，也就是map.get("key1") 将返回object1。你也可以使用其它对象作为键值：

```
anObject.map = $objectKey1:$objectValue1, $objectKey2:$objectValue2
...
```

Considerations 注意事项

Order Matters 顺序问题

上述 INI 格式和约定非常方便也非常易懂，但它并没有另外一种 text/XML的配置路径强大，通过上述途径进行配置需要知道非常重要的一件事情就是顺序问题！

小心

每一个对象实例以及每一个指定的值都将按照其在 *[main]* 区域中产生的顺序的执行，这些行最终转换为 *JavaBeans* 的 *getter/setter* 方法调用，这些方法按同样的顺序调用。

当你写配置文件的时候要牢记于此。

Overriding Instances 覆盖实例

每一个对象都可以被后定义的新实例覆盖，例如，第二个myRealm定义将重写第一个：

```
...
myRealm = com.company.security.MyRealm
...
myRealm = com.company.security.DatabaseRealm
...
```

这样的结果是 myRealm 是 com.company.security.DatabaseRealm 实例而前面的实例不会被使用（会作为垃圾回收）。

Default SecurityManager 默认Default SecurityManager

你可能注意到在以上所有例子中都没有定义 SecurityManager，而我们直接设置其嵌套属性

```
myRealm = ...

securityManager.sessionManager.globalSessionTimeout = 1800000
...
```

这是因为securityManager实例是特殊的--它已经为你实例化过了并且准备好了，所以你并不需要知道指定的实例化SecurityManager的实现类。

当然，如果你确实想指定你自己的实现类，你可以像上面的覆盖实例那样定义你自己的实现：

```
...
securityManager = com.company.security.shiro.MyCustomSecurityManager
...
```

当然，很少需要这样--Shiro 的 SecurityManager 实现可以按需求进行定制，你可能要问一下自己（或者用户群）你是否真的需要这样做。

[users]

[users]区域允许你定义一组静态的用户帐号，这对于那些只有少数用户帐号并且用户帐号不需要在运行时动态创建的环境来说非常有用。下面是一个例子：

```
[users]
admin = secret
lonestarr = vespa, goodguy, schwartz
darkhelmet = ludicrousspeed, badguy, schwartz
```

自动生成*IniRealm*

定义非空的[users]或[roles]区域将自动创建org.apache.shiro.realm.text.IniRealm 实例,在[main]区域下生成一个可用的 iniRealm ，你可以像上面配置其它对象那样配置它。

Line Format 格式

[users]区域下每一行必须和下面的形式一致：

```
username = password, roleName1, roleName2, ..., roleNameN
```

- 等号左边的值是用户名；
- 等号右侧第一个值是用户密码，密码是必须的；
- 密码之后用逗号分割的值是赋予用户的角色名，角色名是可选的。

Encrypting Passwords 密码加密

如果你不希望[users]区域下的密码以明文显示，你可以用你喜欢的哈希算法（MD5, Sha1, Sha256, 等）来加密它们，将加密后的字符串作为密码值，默认的，密码建议用16位编码算法，但也可以用64位编码算法替代（如下）

简单的安全密码

为了节约时间获得最佳实践，你可以使用 *Shiro* 的 *Command Line Hasher*，它可以加密密码和其它类型的资源，尤其使给 *INI[user]* 密码加密变得非常简单。

一旦你指定了加密后的密码值，你必须告诉 shiro 它们是加密的，你可以通过配置配置在[main]隐含创建的iniRealm相应的CredentialsMatcher 实现来告知你使用的哈希算法：

```
[main]
...
sha256Matcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
...
iniRealm.credentialsMatcher = $sha256Matcher
...

[users]
# user1 = sha256-hashed-hex-encoded password, role1, role2, ...
user1 = 2bb80d537b1da3e38bd30361aa855686bde0eacd7162fef6a25fe97bf527a25b, role1, role2
, ...
```

你可以像配置其他对象那样配置 CredentialsMatcher 的所有属性，例如，指定使用salting或者有多少hash iterations执行，可以查看org.apache.shiro.authc.credential.HashedCredentialsMatcher Java文档更好地理解 hashing 策略，可能会很有帮助。

例如，如果你用64位编码方式取代了16位编码方式，你应该指定：

```
[main]
...
# true = hex, false = base64:
sha256Matcher.storedCredentialsHexEncoded = false
```

[roles]

[roles]区域允许你将权限和在[users]定义的角色对应起来，同样的，这对于那些只有少数用户帐号并且用户帐号不需要在运行时动态创建的环境来说非常有用。下面是一个例子：

```
[roles]
# 'admin' role has all permissions, indicated by the wildcard '*'
admin = *
# The 'schwartz' role can do anything (*) with any lightsaber:
schwartz = lightsaber:*
# The 'goodguy' role is allowed to 'drive' (action) the winnebago (type) with
# license plate 'eagle5' (instance specific id)
goodguy = winnebago:drive:eagle5
```

Line Format 格式

[roles]区域下的每一行必须用下面的格式定义角色-权限的键/值对应关系。

```
rolename = permissionDefinition1, permissionDefinition2, ..., permissionDefinitionN
```

权限定义可以是非常随意的字符串，但大部分用户还是希望使用易用而灵活的和 org.apache.shiro.authz.permission.WildcardPermission形式一致的字符串格式。查看 Permissions 文档获取更多关于权限的信息和你可以如何利用它为你服务。

内部用法

注意如果一个特定的权限定义需要用到逗号分隔（如：*printer:5thFloor:print,info*），你需要将该定义用双引号括起来从而避免出错：*"printer:5thFloor:print,info"*。

没有权限的角色

如果你有不需要权限的角色，不需要将它们列入*[roles]*区域，仅仅在 *[users]*区域定义角色名就可以创建它们（如果它们尚不存在）。

[urls]

该区域选项将在Web章节讨论。

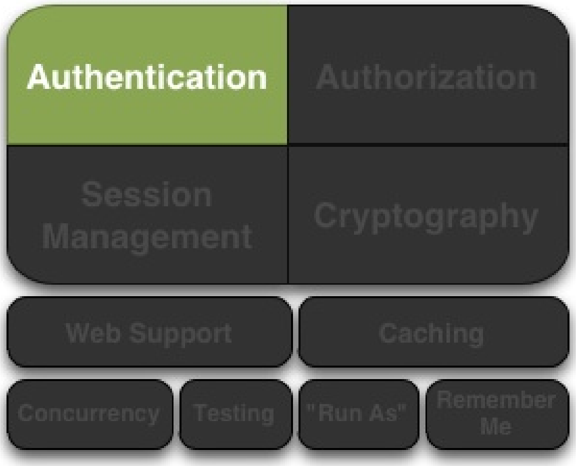
为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

5. Authentication 认证



认证（Authentication）：身份验证的过程--也就是证明一个用户的真实身份。为了证明用户身份，需要提供系统理解和相信的身份信息和证据。

需要通过向 Shiro 提供用户的身份（principals）和证明（credentials ）来判定是否和系统所要求的匹配。

- **Principals(身份)** 是Subject的“标识属性”，可以是任何与Subject相关的标识，比如说名称（给定名称）、名字（姓或者昵称）、用户名、安全号码等等，当然像昵称这样的内容不能很好的对Subject进行独特标识，所以最好的身份信息（Principals）是使用在程序中唯一的标识--典型的使用用户名或邮件地址。
    - **Primary Principal(最主要的身份)**虽然 Shiro 可以使用任何数量的身份，Shiro 还是希望一个程序精确地使用一个主要的身份--一个仅有的唯一标识 Subject 值。在多数程序中经常会是一个用户名、邮件地址或者全局唯一的用户 ID。
- **Credentials(证明)** 通常是只有 Subject 知道的机密内容，用来证明他们真正拥有所需的身份，一些简单的证书例子如密码、指纹、眼底扫描和X.509证书等。

最常见的身份/证明是用户名和密码，用户名是所需的身份说明，密码是证明身份的证据。如果一个提交的密码和系统要求的一致，程序就认为该用户身份正确，因为其他人不应该知道同样的密码。

Authenticating Subjects

Subject 验证的过程可以有效地划分分以下三个步骤：

1.收集 Subject 提交的身份和证明；

2.向 Authentication 提交身份和证明；

3.如果提交的内容正确，允许访问，否则重新尝试验证或阻止访问

下面的代码示范了 Shiro API 如何实现这些步骤：

第一步：收集用户身份和证明

```
//最常用的情况是 username/password 对:
UsernamePasswordToken token = new UsernamePasswordToken(username, password);

//"Remember Me" 功能是内建的
token.setRememberMe(true);
```

在这里我们使用 UsernamePasswordToken，支持所有常用的用户名/密码验证途径，这是一个 org.apache.shiro.authc.AuthenticationToken 接口的实现，这个接口被 Shiro 认证系统用来提交身份和证明。

注意 Shiro 并不关心你如何获取这些信息：也许是用户从一个HTML表单中提交的，或者可能从一个 HTTP 请求字串中解析的，也可能来自于Swing或者 Flex GUI 的密码表单，或者通过命令行参数得到。从程序终端用户获取信息的过程与 Shiro 的 AuthenticationToken 完全无关。

你可以随自己喜欢构造和引用 AuthenticationToken 实例 -- 这是协议无关的。

这个例子同样显示我们希望 Shiro 在尝试验证时执行 “Remember Me” 服务，这确保 Shiro 在用户今后返回系统时能记住他们的身份，我们会在以后的章节讨论 “Remember Me” 服务。

第二步：提交身份和证明

当身份和证明住处被收集并实例化为一个 AuthenticationToken （认证令牌）后，我们需要向 Shiro 提交令牌以执行真正的验证尝试：

```
Subject currentUser = SecurityUtils.getSubject();

currentUser.login(token);
```

在获取当前执行的 Subject 后，我们执行一个单独的 login) 命令，将之前创建的 AuthenticationToken 实例传给它。

调用 login 方法将有效地执行身份验证。

三步：处理成功或失败

当login函数没有返回信息时表明验证通过了。程序可以继续运行，此时执行SecurityUtils.getSubject() 将返回验证后的 Subject 实例，subject.isAuthenticated()) 将返回true。

但是如果 login 失败了呢？例如，用户提供了一个错误的密码或者因访问系统次数过多而被锁定将会怎样呢？

Shiro拥有丰富的运行期异常AuthenticationException可以精确标明为何验证失败，你可以将login 放入到 try/catch 块中并捕获所有你想捕获的异常并对它们做出处理。例如：

```
try {
    currentUser.login(token);
} catch ( UnknownAccountException uae ) { ...
} catch ( IncorrectCredentialsException ice ) { ...
} catch ( LockedAccountException lae ) { ...
} catch ( ExcessiveAttemptsException eae ) { ...
} ... 捕获你自己的异常 ...
} catch ( AuthenticationException ae ) {
    //未预计的错误?
}

//没问题，继续
```

如果原有的异常不能满足你的需求，可以创建自定义的AuthenticationExceptions 来表示特定的失败场景。

登录失败小贴士

虽然你的代码可以对指定的异常做出处理并执行某些所需的逻辑，但有经验的安全做法是仅向终端用户输出一般的失败信息，例如"错误的用户名和密码"。这确保不向尝试攻击你的黑客提供有用的信息。

Remembered vs. Authenticated

如上例所示，Shiro 支持在登录过程中执行"remember me"，在此值得指出，一个已记住的Subject（remembered Subject）和一个正常通过认证的 Subject（authenticated Subject）在 Shiro 是完全不同的。

- 记住的（**Remembered**）：一个被记住的 Subject 没有已知身份（也就是说subject.getPrincipals())返回空），但是它的身份被先前的认证过程记住，并存于先前session中，一个被认为记住的对象在执行subject.isRemembered())返回真。
- 已验证（**Authenticated**）：一个被验证的 Subject 是成功验证后（如登录成功）并存于当前 session 中，一个被认为验证过的对象调用subject.isAuthenticated()) 将返回真。

互斥的

已记住（*Remembered*）和已验证（*Authenticated*）是互斥的--一个标识值为眞另一个就为假，反过来也一样。

Why the distinction?为什么区分？

验证（authentication）有明显的证明含义，也就是说，需要担保Subject 已经被证明他们是谁。

当一个用户仅仅在上一次与程序交互时被记住，证明的状态已经不存在了：被记住的身份只是给系统一个信息这个用户可能是谁，但不确定，没有办法担保这个被记住的 Subject 是所要求的用户，一旦这个 Subject 被验证通过，他们将不再被认为是记住的因为他们的身份已经被验证并存于当前的session中。

所以尽管程序大部分情况下仍可以针对记住的身份执行用户特定的逻辑，比如说自定义的视图，但不要执行敏感的操作直到用户成功执行身份验证使其身份得到确定。

例如，检查一个 Subject 是否可以访问金融信息应该取决于是否被验证（isAuthenticated()）而不是被记住（isRemembered()），要确保该Subject 是所需的和通过身份验证的。

An illustrating example 一个例子说明

下面是一个非常常见的场景帮助说明被记住和被验证之间差别为何重要。

假设你使用Amazon.com，你已经成功登录并且在购物篮中添加了一些书籍，但你由于临时要参加一个会议，匆忙中你忘记退出登录，当会议结束，回家的时间到了，于是你离开了办公室。

第二天当你回到工作，你意识到你没有完成你的购买动作，于是你回到Amazon.com，这时，Amazon.com“记得”是是，通过你的名字向你打招呼，仍旧给你提供个性化的图书推荐，对于Amazon.com，subject.isRemembered() 将返回眞。

但是当你想访问你帐号的信用卡信息完成图书购买的时候会怎样呢？虽然Amazon.com “记住”了你（isRemembered() == true），它不能担保你就是你（也许是正在使用你计算机的同事）。

于是，在你执行像使用信用卡信息之类的敏感操作之前，Amazon.com 强制你登录以使他们担保你的身份，在你登录之后，你的身份已经被验证，对于Amazon.com，isAuthenticated() 将返回眞。

这类情景经常发生，所以 Shiro 加入了该功能，你可以在你的程序中使用。现在是使用 isRemembered() 还是使用 isAuthenticated() 来定制你的视图和工作流完全取决于你自己，但 Shiro 维护这种状态基础以防你可能会需要。

Logging Out 退出登录

与验证相对的是释放所有已知的身份信息，当 Subject 与程序不再交互了，你可以调用 subject.logout() 丢掉所有身份信息。

```
currentUser.logout(); //清除验证信息，使 session 失效
```

当你调用 logout，任何现存的 session 将变为不可用并且所有的身份信息将消失（如：在 web 程序中，RememberMe 的 Cookie 信息同样被删除）。

当一个 Subject 退出登录，Subject 被重新认定为匿名的，对于 web 程序，如果需要可以重新登录。

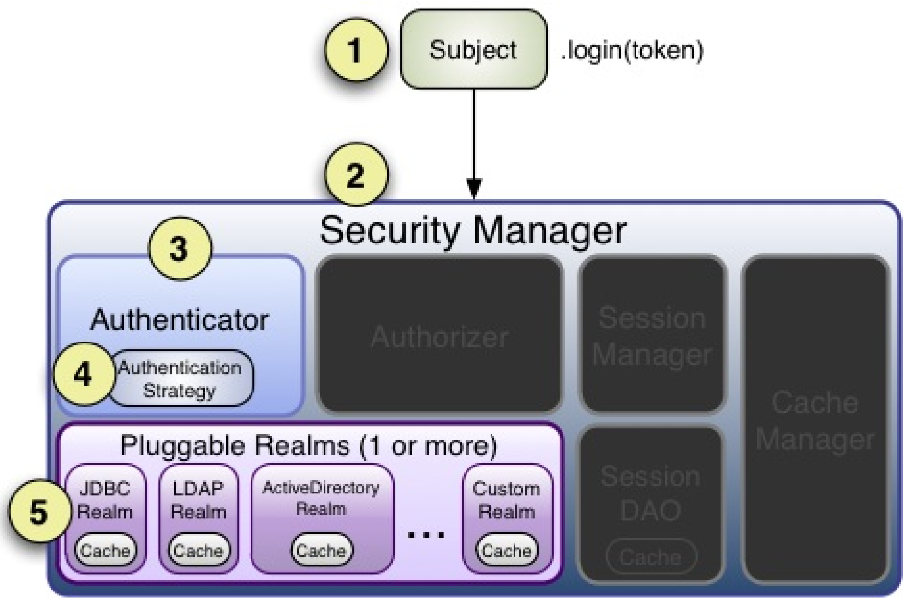
Web 程序需注意

因为在 *Web* 程序中记住身份信息往往使用 *Cookies*，而 *Cookies* 只能在 *Response* 提交时才能被删除，所以强烈要求在为最终用户调用*subject.logout()* 之后立即将用户引导到一个新页面，确保任何与安全相关的 *Cookies* 如期删除，这是 *Http* 本身 *Cookies* 功能的限制而不是 *Shiro* 的限制。

Authentication Sequence 认证序列

直到现在，我们只看到如何在程序代码中验证一个 Suject，现在我们看一下当一个验证发生时 Shiro 内部发生了什么。

我们仍使用之前在架构Architecture章节里见到过的架构图，仅将左侧跟认证相关的组件高亮，每一个数字代表认证中的一个步骤：



第**1**步：程序代码调用 Subject.login 方法，向AuthenticationToken（认证令牌）实例的构造函数传递用户的身份和证明。

第**2**步：Subject 实例，通常是一个 DelegatingSubject（或其子类）通过调用 securityManager.login(token)将这个令牌转交给程序的 SecurityManager。

第**3**步：SecurityManager，基本的“安全伞”组件，得到令牌并通过调用 authenticator.authenticate(token))简单地将其转交它内部的 Authenticator 实例，大部分情况下是一个 ModularRealmAuthenticator 实例，用来支持在验证过程中协调一个或多个Realm实例。ModularRealmAuthenticator 本质上为 Apache Shiro（在 PAM 术语中每一个 Realm 称为一个“模块”）提供一个 PAM 类型的范例。

第**4**步：如程序配置了多个 Realm，ModularRealmAuthenticator实例将使用其配置的 AuthenticationStrategy 开始一个 多 Realm 身份验证的尝试。在 Realm 被验证调用的整个过程中，AuthenticationStrategy（安全策略）被调用用来回应每个Realm结果，我们将稍后讨论 AuthenticationStrategies。

注意：单 *Realm* 程序

如果仅有一个 *Realm* 被配置，它直接被调用--在单 *Realm* 程序中不需要 *AuthenticationStrategy*。

第**5**步：每一个配置的 Realm 都被检验看其是否支持)提交的AuthenticationToken，如果支持，则该 Realm 的 getAuthenticationInfo) 方法随着提交的牌被调用，getAuthenticationInfo 方法为特定的 Realm 有效提供一次独立的验证尝试，我们将稍后讨论 Realm 验证行为。

Authenticator

就像以前提到过的，Shiro SecurityManager implementations默认使用一个 ModularRealmAuthenticator 实例， ModularRealmAuthenticator 同样支持单 Realm 和多 Realm。

在一个单 Realm 程序中，ModularRealmAuthenticator 将直接执行单独的 Realm，如果配置有两个或以上 Realm，将会使用AuthenticationStrategy 实例来协调如何进行验证，我们将在下面的章节中讨论 AuthenticationStrategy。

如果你希望用自定义的 Authenticator 实现配置 SecurityManager，你可以在 shiro.ini 中做这件事，如：

```
[main]
...
authenticator = com.foo.bar.CustomAuthenticator

securityManager.authenticator = $authenticator
```

尽管在实际操作中， ModularRealmAuthenticator 适用于大部分需求。

AuthenticationStrategy

当一个程序中定义了两个或多个 realm 时，ModularRealmAuthenticator 使用一个内部的 AuthenticationStrategy 组件来决定一个验证是否成功。

例如，如果一个 Realm 验证一个 AuthenticationToken 成功，但其他的都失败了，那这次尝试是否被认为是成功的呢？是不是所有 Realm 验证都成功了才认为是成功？又或者一个 Realm 验证成功，是否还有必要讨论其他Realm？AuthenticationStrategy 根据程序需求做出恰当的决定。

AuthenticationStrategy 是一个 stateless 的组件，在整个验证过程中在被用到4次（在这4次活动中需要必要的 state 时，state 将作为方法参数传递）

1.在任何 Realms 被执行之前；

2.在某个的 Realm 的 getAuthenticationInfo 方法调用之前；

3.在某个的 Realm 的 getAuthenticationInfo 方法调用之后；

4.在所有的 Realm 被执行之后。

AuthenticationStrategy 还有责任从每一个成功的 Realm 中收集结果并将它们“绑定”到一个单独的 AuthenticationInfo，这个AuthenticationInfo 实例是被 Authenticator 实例返回的，并且 shiro 用它来展现一个 Subject 的最终身份（也就是 Principals ）。

Subject 身份“展示（*view*）”

如果你在程序中使用多于一个的 *Realm* 从多个数据源中获取帐户数据，程序可看到的是 *AuthenticationStrategy* 最终负责 *Subject* 身份最终“合并（*merged*）”的视图

Shiro 有3个具体的 AuthenticationStrategy 实现：

`AuthenticationStrategy` class	Description
`AtLeastOneSuccessfulStrategy`	如果有一个或多个Realm验证成功，所有的尝试都被认为是成功的，如果没有一个验证成功，则该次尝试失败
`FirstSuccessfulStrategy`	只有从第一个成功验证的Realm返回的信息会被使用，以后的Realm将被忽略，如果没有一个验证成功，则该次尝试失败
`AllSuccessfulStrategy`	所有配置的Realm在全部尝试中都成功验证才被认为是成功，如果有一个验证不成功，则该次尝试失败。

ModularRealmAuthenticator 默认使用 AtLeastOneSuccessfulStrategy 实现，这也是最常用的策略，然而你也可以配置你希望的不同的策略。

shiro.ini

```
[main]
...
authcStrategy = org.apache.shiro.authc.pam.FirstSuccessfulStrategy

securityManager.authenticator.authenticationStrategy = $authcStrategy

...
```

自定义的 *AuthenticationStrategy*

如果你希望创建你自己的 *AuthenticationStrategy* 实现，你可以使用 *org.apache.shiro.authc.pam.AbstractAuthenticationStrategy* 作为起始点。*AbstractAuthenticationStrategy* 类自动实现 *'*绑定（*bundling*）*'/*聚集（*aggregation*）行为将来自于每个*Realm* 的结果收集到一个 *AuthenticationInfo* 实例中。

Realm 验证的顺序

非常重要的一点是，和 Realm 交互的 ModularRealmAuthenticator 按迭代（iteration）顺序执行。

ModularRealmAuthenticator 可以访问为 SecurityManager 配置的 Realm 实例，当尝试一次验证时，它将在集合中遍历，支持对提交的 AuthenticationToken 处理的每个 Realm 都将执行 Realm 的 getAuthenticationInfo 方法。

隐含的顺序

在使用 ShiroINI 配置文件形式时，你可以按你希望其处理 AuthenticationToken 的顺序来配置 Realm，例如，在shiro.ni 中，Realm 将按照他们在INI文件中定义的顺序执行。

```
blahRealm = com.company.blah.Realm
...
fooRealm = com.company.foo.Realm
...
barRealm = com.company.another.Realm
```

SecurityManager上配置了这三个 Realm，在一个验证过程中，blahRealm, fooRealm, 和 barRealm 将被顺序执行。

这基本上与定义下面这一行语句的效果相同：

```
securityManager.realms = $blahRealm, $fooRealm, $barRealm
```

使用这种方法，你不需要设置 securityManager 的 realm 顺序，每一个被定义的realm 将自动加入到 realms 属性中。

Explicit Ordering明确的顺序

如果你希望明确定义 realm 执行的顺序，不管他们如何被定义，你可以设置 SecurityManager 的 realms 属性，例如，使用上面定义的 realm，但你希望 blahRealm 最后执行而不是第一个：

```
blahRealm = com.company.blah.Realm
...
fooRealm = com.company.foo.Realm
...
barRealm = com.company.another.Realm

securityManager.realms = $fooRealm, $barRealm, $blahRealm
```

明确 *Realm* 包含

当你明确的配置 *securityManager.realms* 属性时，只有被引用的 *realm* 将为 *SecurityManager* 配置，也就是说你可能在 *INI* 中定义了*5*个 *realm*，但实际上只使用了*3*个，如果在 *realm* 属性中只引用了*3*个，这和隐含的 *realm* 顺序不同，在那种情况下，所有有效的 *realm* 都会用到。

Realm Authentication 验证

本章阐述了当一个验证尝试发生时 Shiro 主要的工作流程，而在验证过程中，用到的 Realm 内产生的工作流程（如上面提到的第5步）将在 Realm 章中 Realm Authentication 节讨论。

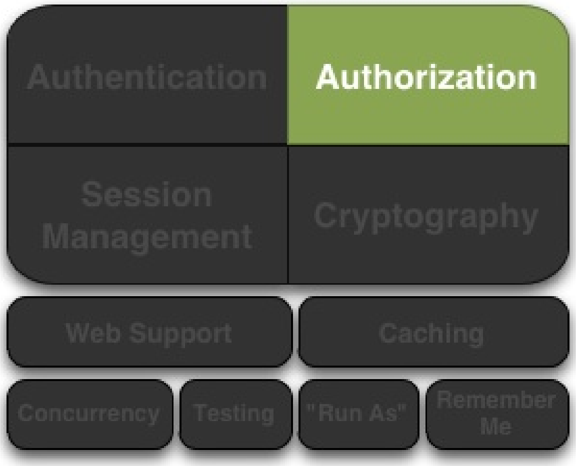
为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

6. Authorization 授权



授权，亦即访问控制，是管理资源访问的过程，换言之，也就是控制在一个程序中“谁”有权利访问“什么”。

授权的例子有：是否允许这个用户查看这个页面，编辑数据，看到按钮，或者从这台打印机打印？这些决定是一个用户可以访问什么的决断。

Elements of Authorization 元素

授权有三个核心元素，在 Shiro 中我们经常要用到它们：权限（permissions）、角色（roles）和用户（users）。

Permissions

权限是 Apache Shiro 中安全策略最基本的元素，它们是一组关于行为的基本指令，以明确表示在一个程序中什么可以做。一个很好定义的权限指令必须描述资源以及当一个 Subject 与这些资源交互时什么动作可以执行。

下面是一些权限指令的例子：

- 打开一个文件；
- 查看“/user/list”页面；
- 打印文档；
- 删除“Jsmith”用户。

大部分资源都支持基本的 CRUD（create,read,update,delete）操作，但对于特定资源类型，任何动作都是可以的。权限设置最基础的思想是在资源和动作的基础上设置最小量的权限指令。

当看到权限时，最重要的一点是要是认识到一个权限指令不是描述“谁”可以执行这个动作，而是描述“什么”可以做的指令。

权限只描述行为

权限指令只描述行为（和资源相关的动作），并不关心“谁”有能力执行这个动作。

定义“谁”（用户）被允许做“什么”（权限）需要用一些方法将权限赋给用户，这通常取决于程序的数据模型而且经常在程序中发生改变。

例如，一组权限可以归于一个角色而角色与一个或多个用户对象关联，或者一些程序可以有一组用户而一个组可以指定一个角色，在这里关系将被传递也就是说组内用户隐含被赋予角色的权限。

有很多方式可以将权限赋予用户--程序根据需求决定如何设计。

我们稍后讨论 Shiro 如何判断一个 Subject 是否被允许。

Permission Granularity 权限粒度

上面的权限示例都是针对资源（门、文件、客户等）指定的动作（打开、读、删除等），在一些场景中，他们也会指定非常细粒度的“实例级别”行为--例如,“删除”（delete）名为“Jsmith”（实例标识）的“用户”（资源类型），在 Shiro 中，你可以精确定义指令到你所能细化到的程度。

我们在 Shiro 的 Permissions 文档中详细讨论权限粒度和权限指令的“级别”。

Roles

角色是一个实体名，代表一组行为或职责，这些行为在程序中转换为你可以或者不能做的事情。角色通常赋给用户帐号，关联后，用户就可以“做”属于不同角色的事情。

有两种有效的角色指定方式，Shiro 都支持。

- 权限隐含于角色中：大部分用户将权限隐含于创建的角色中：程序只是在一个角色名称上隐含了一组行为（也就是权限），使用时，在软件级别不会说“某角色允许执行行为A、B和C”，而是将行为隐含于一个单独的角色名字中。

潜在的安全隐患

虽然这是一个非常简单和常用的方法，但隐含的角色可能会增加软件的维护成本和管理问题。

例如，如果你想增加或删除一个角色，或者重定义角色的行为怎么办？你不得不重新打开代码，修改所有对更改后的角色的检测，每次都需要这样做，这还没提到其引起的执行代价（重测试，通过质量验证，关闭程序，升级软件，重启程序等）。

对于简单程序这种方法可能适用（比如只有一个*'admin'*角色和*'everyone else'*角色），但复杂的程序中，这会成为你程序生命周期中一个主要的问题，会给你的软件带来很大的维护代价。

- 明确为角色指定权限：明确为角色指定权限本质上是一组权限指令的名称集，在这种形式下，程序（以及 Shiro）准确知道一个特定的角色是什么意思，因为它确切知道某行为是否可以执行，而不用去猜测特定的角色可以或不可以做什么。

Shiro 团队提倡使用权限和明确为角色指定权限替代原始的将权限隐含于角色中的方法，这样你可以对程序安全提供更强的控制。

基于资源的访问控制

读一下这个文章：*新的RBAC：基于资源的权限管理(Resource-Based Access Control)*，这篇文章深入讨论了使用权限和明确为角色指定权限代替旧的将权限隐含于角色中方法的好处（以及对源代码的影响）。

Users用户

一个用户本质上是程序中的“谁”，如同我们前面提到的，Subject 实际上是 shiro 的“用户”。

用户（Subjects）通过与角色或权限关联确定是否被允许执行程序内特定的动作，程序数据模型确切定义了 Subject 是否允许做什么事情。

例如，在你的数据模型中，你定义了一个普通的用户类并且直接为其设置了权限，或者你只是直接给角色设置了权限，然后将用户与该角色关联，通过这种关联，用户就“有”了角色所具备的权限，或者你也可以通过“组”的概念完成这件事，这取决于你程序的设计。

数据模型定义了如何进行授权，Shiro 依赖一个 Realm 实现将你的数据模型关联转换成 Shiro 可以理解的内容，我们将稍后讨论 Realms。

最终，是 *Realm* 与你的数据源（*RDBMS,LDAP*等）做交流，*Realm* 用来告知*Shiro* 是否角色或权限是否存在，你可以完全控制你的授权模型如何创建和定义。

Authorizing Subjects 授权对象

在 Shiro 中执行授权可以有三种途径：

- 程序代码--你可以在你的 JAVA 代码中执行用类似于 if 和 else 的结构来执行权限检查。
- JDK 注解--你可以在你的 JAVA 方法上附加权限注解
- JSP/GSP 标签--你可以基于角色和权限控制 JSP 或 GSP 页面输出内容。

Programmatic Authorization 程序中检查授权

直接在程序中为当前 Subject 实例检查授权可能是最简单也最常用的方法。

Role-Based Authorization 基于角色的授权

如果你要基于简单/传统的角色名进行访问控制，你可以执行角色检查：

Role Checks 角色检查

如果你想简单地检查一下当前Subject是否拥有一个角色，你可以在一个实例上调用 hasRole* 方法。

例如，查看一个 Subject 是否有特定（单独）的角色，你可以调用 subject.hasRole(roleName))方法，做出相应的反馈。

```
Subject currentUser = SecurityUtils.getSubject();

if (currentUser.hasRole("administrator")) {
    //显示 admin 按钮
} else {
    //不显示按钮?  灰色吗？
}
```

下面是你可以根据需要调用的函数：

Subject 方法	描述
hasRole(String roleName)	如果Subject指定了特定的角色返回真，否则返回假；
hasRoles(List<String> roleNames)	返回一个与参数顺序相对应的hasRole结果数组，当一次有多个角色需要检测时非常有用（如定制一个复杂的视图）
hasAllRoles(Collection<String> roleNames)	如果Subject具备所有角色返回真，否则返回假。

Role Assertions 角色判断

还有另一个方法检测 Subjet 是否是指定为某个角色，你可以在的代码执行之前简单判断他们是否是所要求的角色，如果 Subject 不是所要求的角色， AuthorizationException 异常将被抛出，如果是所要求的角色，判断将安静地执行并按期望顺序执行下面的逻辑。

例如：

```
Subject currentUser = SecurityUtils.getSubject();

//保证当前用户是一个银行出纳员
//因此允许开立帐户：
currentUser.checkRole("bankTeller");
openBankAccount();
```

与 hasRole* 方法相比，这种方法的好处在于代码更为清晰，如果当前Subject 不满足所需条件你不需要建立你自己的AuthorizationExceptions 异常（如果你不想那么做）。

下面是你可以根据需要调用的函数：

Subject 方法	描述
`checkRole(String roleName)`	如果Subject被指定为特定角色则安静地返回否则抛出AuthorizationException异常；
`checkRoles(Collection roleNames)`	如果Subject被指定了所有特定的角色则安静地返回否则抛出AuthorizationException异常；
`checkRoles(String... roleNames)`	和上面的checkRoles具有相同的效果，但允许Java5的变参形式。

Permission-Based Authorization 基于权限的授权

就像我们上面在角色概述中提到的，通过基于权限的授权执行访问控制是更好的方法。基于权限的授权，因为其与程序功能（以及程序核心资源上的行为）紧密联系，基于权限授权的源代码在程序功能改变时才需要改变，而与安全策略无关。这意味着与同样基于角色的授权相比，对代码的影响更少。

Permission Checks 权限检查

如果你希望检查一个 Subject 是否允许做某件事情，你可以调用isPermitted* 方法的变形，有两种主要方式检查授权--基于对象的权限实例和基于字符串的权限表示。

Object-based Permission Checks 基于对象的权限检查

执行权限检查的一种方法是实例化一个Shiro的org.apache.shiro.authz.Permission接口并且将它传递给接收权限实例的*isPermitted 方法。

例如，假设一下以下情景：办公室里有一台唯一标识为 laserjet4400n 的打印机，在我们向用户显示打印按钮之前，软件需要检查当前用户是否允许用这台打印机打印文档，检查权限的方式会是这样：

```
Permission printPermission = new PrinterPermission("laserjet4400n", "print");

Subject currentUser = SecurityUtils.getSubject();

if (currentUser.isPermitted(printPermission)) {
    //显示 打印 按钮
} else {
    //不显示按钮?  灰色吗？
}
```

在这个例子中，我们同样看到了一个非常强大的实例级别的访问控制检查--在单独数据实例上限制行为的能力。

基于对象的权限对下列情况非常有用：

- 希望编译期类型安全；
- 希望确保正确地引用和使用的权限；
- 希望对权限判断逻辑（称作权限隐含逻辑，基于权限接口的 implies)方法）执行方式进行明确控制；
- 希望确保权限正确地反映程序资源（例如，在一个对象域模型上创建一个对象时，权限类可能自动产生）。

下面是你可以根据需要调用的函数：

Subject 方法	描述
isPermitted(Permission p)	如果Subject允许执行特定权限实例综合指定的动作或资源访问权返回眞，否则返回假；
isPermitted(List perms)	按参数顺序返回isPermitted的结果数组，如果许多权限需要检查时非常有用（如定制一个复杂的视图）
isPermittedAll(Collection perms)	如果Subject拥有指定的所有权限返回眞，否则返回假。

String-based permission checks 基于字符串的权限检查

虽然基于对象的权限检查很有用（编译期类型安全，对行为担保，定制隐含逻辑等），但在许多程序里有时候感觉有点笨重，另一种选择是用普通的字符串来代表权限。

例如，对于上面打印权限的例子，我们可以使用字符串权限检查达到同样的结果

```
Subject currentUser = SecurityUtils.getSubject();

if (currentUser.isPermitted("printer:print:laserjet4400n")) {
    //显示 打印 按钮
} else {
    //不显示按钮?  灰色吗?
}
```

这个例子同样实现了实例级别的权限检查，但是所有主要权限部件--printer（资源类型）、print（动作）、laserjet4400n（实例ID）都表现为一个字符串。

上面的例子展示了一种以冒号分割的特殊形式的字符串，定义于Shiro的org.apache.shiro.authz.permission.WildcardPermission中，它适合大多数用户的需求。

上面的代码块基本上是下面这段代码的缩写：

```
Subject currentUser = SecurityUtils.getSubject();

Permission p = new WildcardPermission("printer:print:laserjet4400n");

if (currentUser.isPermitted(p) {
    //显示 打印 按钮
} else {
    //不显示按钮?  灰色吗?
}
```

WildcardPermission 令牌形式和构成选项将在 Shiro 的 Permission文档中深入讨论

上面的字符串使用默认的 WildcardPermission 格式，实际上你可以创造并使用你自己的字符串格式，我们将在下面 Realm 授权章节讨论如何这样做。

基于字符串的权限有利的一面在于你不需要实现一个接口而且简单的字符串也非常易读，而不利的一面在于不保证类型安全，而且当你需要定义超出字符串表现能力之外的更复杂的行为时，你仍旧需要利用权限接口实现你自己的权限对象。实际上，大部分 Shiro 的终端用户因为其简单而选择基于字符串的方式，但最终你的程序需求决定了哪一种方法会更好。

和基于对象的权限检查方法一样，下面是字符串权限检查的函数：

Subject 方法	描述
isPermitted(String perm)	如果Subject被允许执行字符串表达的动作或资源访问权限，返回真，否则返回假；
isPermitted(String... perms)	按照参数顺序返回isPermitted的结果数组，当许多字符串权限需要检查时非常有用（如定制一个复杂的视图时）；
isPermittedAll(String... perms)	当Subject具备所有字符串定义的权限时返回真，否则返回假。

Permission Assertions 权限判断

另一种检查 Subject 是否被允许做某件事的方法是，在逻辑执行之前简单判断他们是否具备所需的权限，如果不允许，AuthorizationException异常被抛出，如果是允许的，判断将安静地执行并按期望顺序执行下面的逻辑。

例如：

```
Subject currentUser = SecurityUtils.getSubject();

//担保允许当前用户
//开一个银行帐户：
Permission p = new AccountPermission("open");
currentUser.checkPermission(p);
openBankAccount();
```

或者，同样的判断，可以用字符串形式：

```
Subject currentUser = SecurityUtils.getSubject();

//担保允许当前用户
//开一个银行帐户：
currentUser.checkPermission("account:open");
openBankAccount();
```

与 isPermitted* 方法相比较，这种方法的优势是代码更为清晰，如果当前Subject 不符合条件，你不必创建你自己的 AuthorizationExceptions 异常（如果你不想那么做）。

下面是你可以根据需要调用的函数：

Subject 方法	描述
`checkPermission(Permission p)`	如果Subject被允许执行特定权限实例指定的动作或资源访问，安静地返回，否则抛出AuthorizationException异常。
`checkPermission(String perm)`	如果Subject被允许执行权限字符串指定的动作或资源访问，安静地返回，否则抛出AuthorizationException异常。
`checkPermissions(Collection perms)`	如果Subject被允许执行所有权限实例指定的动作或资源访问，安静地返回，否则抛出AuthorizationException异常。
`checkPermissions(String... perms)`	和上面的checkPermissions效果一样，只是使用字符串权限类型。

Annotation-based Authorization 基于注解的授权

如果你更喜欢基于注解的授权控制，除了 Subject 的 API 之外，Shiro提供了一个 Java 5 的注解集。

Configuration 配置

在你使用 JAVA 的注解之前，你需要在程序中启动 AOP 支持，因为有许多AOP 框架，所以很不幸，在这里并没有标准的在程序中启用 AOP 的方法。

关于AspectJ，你可以查看我们的AspectJ sample application；

关于Spring，你可以查看 Spring Integration文档；

关于Guice，你可以查看我们的 Guice Integration文档；

RequiresAuthentication 注解

RequiresAuthentication 注解表示在访问或调用被注解的类/实例/方法时，要求 Subject 在当前的 session中已经被验证。

举例：

```
@RequiresAuthentication
public void updateAccount(Account userAccount) {
    //这个方法只会被调用在
    //Subject 保证被认证的情况下
    ...
}
```

这基本上与下面的基于对象的逻辑效果相同：

```
public void updateAccount(Account userAccount) {
    if (!SecurityUtils.getSubject().isAuthenticated()) {
        throw new AuthorizationException(...);
    }

    //这里 Subject 保证被认证的情况下
    ...
}
```

RequiresGuest 注解

RequiresGuest 注解表示要求当前Subject是一个“guest(访客)”，也就是，在访问或调用被注解的类/实例/方法时，他们没有被认证或者在被前一个Session 记住。

例如：

```
@RequiresGuest
public void signUp(User newUser) {
    //这个方法只会被调用在
    //Subject 未知/匿名的情况下
    ...
}
```

这基本上与下面的基于对象的逻辑效果相同：

```
public void signUp(User newUser) {
    Subject currentUser = SecurityUtils.getSubject();
    PrincipalCollection principals = currentUser.getPrincipals();
    if (principals != null && !principals.isEmpty()) {
        //已知的身份 - 不是 guest（访客）:
        throw new AuthorizationException(...);
    }

    //在这里 Subject 确保是一个 'guest（访客）'
    ...
}
```

RequiresPermissions 注解

RequiresPermissions 注解表示要求当前Subject在执行被注解的方法时具备一个或多个对应的权限。

例如：

```
@RequiresPermissions("account:create")
public void createAccount(Account account) {
    //这个方法只会被调用在
    //Subject 允许创建一个 account 的情况下
    ...
}
```

这基本上与下面的基于对象的逻辑效果相同

```
public void createAccount(Account account) {
    Subject currentUser = SecurityUtils.getSubject();
    if (!subject.isPermitted("account:create")) {
        throw new AuthorizationException(...);
    }

    //在这里 Subject 确保是允许
    ...
}
```

RequiresRoles 注解

RequiresRoles 注解表示要求当前Subject在执行被注解的方法时具备所有的角色，否则将抛出 AuthorizationException 异常。

例如：

```
@RequiresRoles("administrator")
public void deleteUser(User user) {
    //这个方法只会被 administrator 调用
    ...
}
```

这基本上与下面的基于对象的逻辑效果相同

```
public void deleteUser(User user) {
    Subject currentUser = SecurityUtils.getSubject();
    if (!subject.hasRole("administrator")) {
        throw new AuthorizationException(...);
    }

    //Subject 确保是一个 'administrator'
    ...
}
```

RequiresUser 注解

RequiresUser* 注解表示要求在访问或调用被注解的类/实例/方法时，当前 Subject 是一个程序用户，“程序用户”是一个已知身份的 Subject，或者在当前 Session 中被验证过或者在以前的 Session 中被记住过。

例如：

```
@RequiresUser
public void updateAccount(Account account) {
    //这个方法只会被 'user' 调用
    //i.e. Subject 是一个已知的身份with a known identity
    ...
}
```

这基本上与下面的基于对象的逻辑效果相同

```
public void updateAccount(Account account) {
    Subject currentUser = SecurityUtils.getSubject();
    PrincipalCollection principals = currentUser.getPrincipals();
    if (principals == null || principals.isEmpty()) {
        //无身份 - 他们是匿名的，不被允许
        throw new AuthorizationException(...);
    }

    //Subject 确保是一个已知的身份
    ...
}
```

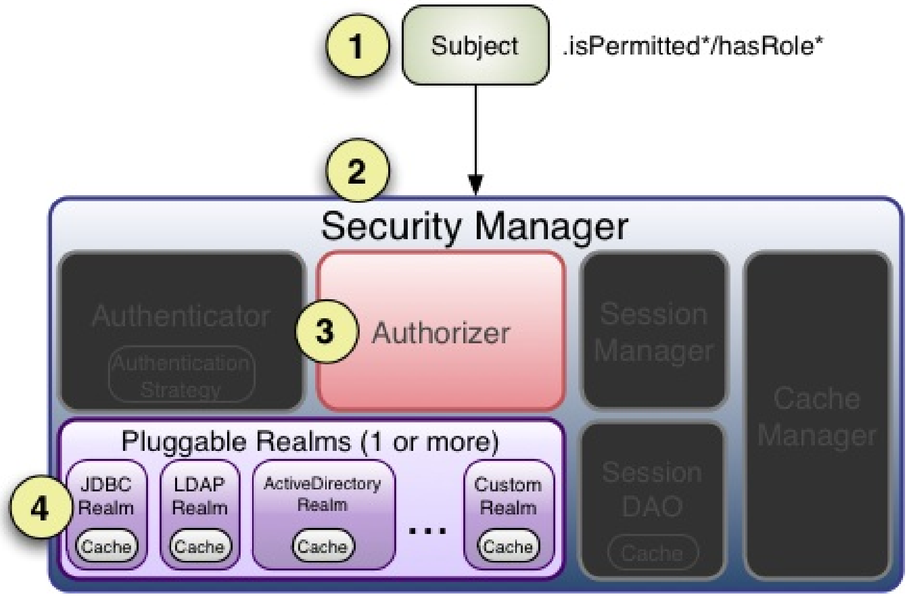
JSP TagLib Authorization 标签库授权

Shiro 提供了一个标签库来控制 JSP/GSP 页面输出，这将在 Web 文档中的 JSP/GSP 标签库中讨论

Authorization Sequence 授权序列

现在我们已经看到如何对当前 Subject 执行授权，让我们了解一下当一个授权命令调用时 Shiro 内部发生了什么事情。

我们仍使用前面Architecture章节里的架构图，在左侧仅仅与授权相关的组件是高亮的，每一个数字代表授权操作中的一个步骤：



第**1**步：程序或框架代码调用一个 Subject 的 `hasRole*` 、 `checkRole*` 、 `isPermitted*` 或者 `checkPermission*` 方法，传递所需的权限或角色。

第**2**步：Subject实例，通常是一个 DelegatingSubject（或子类），通过调用securityManager 与各 `hasRole*` 、 `checkRole*` 、 `isPermitted*` 或 `checkPermission*` 基本一致的方法将权限或角色传递给程序的 SecurityManager(实现了 org.apache.shiro.authz.Authorizer 接口)

第**3**步：SecurityManager 作为一个基本的"保护伞"组件,接替/代表其内部 org.apache.shiro.authz.Authorizer 实例通过调用 authorizer 的各自的 `hasRole*` , `checkRole*` , `isPermitted*` ,或 `checkPermission*` 方法。 authorizer 默认情况下是一个实例 ModularRealmAuthorizer 支持协调一个或多个实例 Realm 在任何授权操作实例。

第**4**步:，检查每一个被配置的 Realm 是否实现相同的 Authorizer接口，如果是，Realm 自己的各 `hasRole*` 、 `checkRole*` 、 `isPermitted*` 或 `checkPermission*` 方法被调用。

ModularRealmAuthorizer

前面提到过，Shiro SecurityManager 默认使用 ModularRealmAuthorizer 实例，ModularRealmAuthorizer 实例同等支持用一个 Realm 的程序和用多个 Realm 的程序。

对于任何授权操作，ModularRealmAuthorizer 将在其内部的 Realm 集中迭代（iterator），按迭代（iteration）顺序同每一个 Realm 交互，与每一个 Realm 交互的方法如下：

1.如果Realm实现了 Authorizer 接口，调用它各自的授权方法（ `hasRole*` 、 `checkRole*` 、 `isPermitted*` 或 `checkPermission*` ）。

1.1.如果 Realm 函数的结果是一个 exception，该 exception 衍生自一个 Subject 调用者的 AuthorizationException，就切断授权过程，剩余的授权 Realm 将不在执行。

1.2.如果 Realm 的方法是一个 `hasRole*` 或 `isPermitted*` ，并且返回真，则真值立即被返回而且剩余的 Realm 被短路，这种做法作为一种性能增强，在一个 Realm 判断允许后，隐含认为这个 Subject 被允许。它支持最安全的安全策略：默认情况下所有都被禁止，明确指定允许的事情。

2.如果 Realm 没有实现 Authorizer 接口，将被忽略。

Realm Authorization Order 授权顺序

需要指出非常重要的一点，就如同验证（authentication）一样，ModularRealmAuthorizer 按迭代（iteration）顺序与 Realm 交互。

ModularRealmAuthorizer 拥有 SecurityManager 配置的 Realm 实例的入口，当执行一个授权操作时，它将在整个集合中进行迭代（iteration），对于每一个实现 Authorizer 接口的 Realm，调用Realm 各自的 Authorizer 方法（如 hasRole、 *checkRole*、 isPermitted或 *checkPermission*）。

Configuring a global PermissionResolver 配置全局的 PermissionResolver

当执行一个基于字符串的权限检查时，大部分 Shiro 默认的 Realm 将会在执行权限隐含逻辑之前首先把这个字符串转换成一个常用的权限实例。

这是因为权限被认为是基于隐含逻辑而不是相等检查（查看Permission章节了解更多隐含与相等的对比）。隐含逻辑用代码表示要比通过字符串对比好，因此，大部分 Realm需要转换一个提交的权限字符串为对应的权限实例。

为了这个转换目的，Shiro 支持 PermissionResolver，大部分 Shiro Realm 使用 PermissionResolver 来支持它们对Authorizer 接口中基于字符串权限方法的实现：当这些方法在Realm上被调用时，将使用PermissionResolver 将字符串转换为权限实例，并执行检查。

所有的 Shiro Realm 默认使用内部的 WildcardPermissionResolver，它使用 Shiro 的 WildcardPermission字符串格式。

如果你想创建你自己的 PermissionResolver 实现，比如说你想创建你自己的权限字符串语法，希望所有配置的Realm实例都支持这个语法，你可以把自己的 PermissionResolver 设置成全局，供所有 realm 使用。

如，在shiro.ini中：

```
globalPermissionResolver = com.foo.bar.authz.MyPermissionResolver
...
securityManager.authorizer.permissionResolver = $globalPermissionResolver
...
```

PermissionResolverAware

如果你想配置一个全局的 *PermissionResolver*，每一个会读取这个*PermissionResolver* 配置的 *Realm* 必须实现*PermissionResolverAware* 接口，这确保被配置 *PermissionResolver* 的实例可以传递给支持这种配置的每一个 *Realm*。

如果你不想使用一个全局的 PermissionResolver 或者你不想被PermissionResolverAware 接口麻烦，你可以明确地为单个的 Realm 配置 PermissionResolver 接口（可看作是JavaBean 的setPermissionResolver 方法）：

```
permissionResolver = com.foo.bar.authz.MyPermissionResolver

realm = com.foo.bar.realm.MyCustomRealm
realm.permissionResolver = $permissionResolver
...
```

Configuring a global RolePermissionResolver 配置全局的 RolePermissionResolver

与 PermissionResolver 类似，RolePermissionResolver 有能力表示执行权限检查的 Realm 所需的权限实例。

最主要的不同在于接收的字符串是一个角色名，而不是一个权限字符串。

RolePermissionResolver 被 Realm 在需要时用来转换一个角色名到一组明确的权限实例。

这是非常有用的，它支持那些遗留的或者不灵活的没有权限概念的数据源。

例如，许多 LDAP 目录存储角色名称（或组名）但不支持角色名和权限的联合，因为它没有权限的概念。一个使用 shiro 的程序可以使用存储于 LDAP 的角色名，但需要实现一个 RolePermissionResolver 来转换 LDAP 名到一组确切的权限中以执行明确的访问控制，权限的联合将被存储于其它的数据存储中，比如说本地数据库。

因为这种将角色名转换为权限的概念是特定的，Shiro 默认的 Realm 没有使用它们。

然而，如果你想创建你自己的 RolePermissionResolver 并且希望用它配置多个 Realm 实现，你可以将你的 RolePermissionResolver设置成全局。

shiro.ini

```
globalRolePermissionResolver = com.foo.bar.authz.MyPermissionResolver
...
securityManager.authorizer.rolePermissionResolver = $globalRolePermissionResolver
...
```

RolePermissionResolverAware

如果你想配置一个全局的 *RolePermissionResolver,* 每个 *Realm* 接收必须实现了 *RolePermisionResolverAware* 接口的配置了的 *RolePermissionResolver* 。这保证了配置全局 *RolePermissionResolver* 实例可以传递到各个支持这样配置的 *Realm* 。

如果你不想使用全局的 RolePermissionResolver 或者你不想麻烦实现 RolePermissionResolverAware 接口，你可以单独为一个 Realm 配置 RolePermissionResolver（可以看作 JavaBean 的 setRolePermissionResolver 方法）。

```
rolePermissionResolver = com.foo.bar.authz.MyRolePermissionResolver

realm = com.foo.bar.realm.MyCustomRealm
realm.rolePermissionResolver = $rolePermissionResolver
...
```

Custom Authorizer 定制Authorizer

如果你的程序使用多于一个 Realm 来执行授权而 ModularRealmAuthorizer 默认的简单迭代（iteration）、短路授权的行为不能满足你的需求，你可以创建自己的 Authorizer 并配置给相应的 SecurityManager。

例如，在shiro.ini中：

```
[main]
...
authorizer = com.foo.bar.authz.CustomAuthorizer

securityManager.authorizer = $authorizer
```

6.1. Permissions 权限

Shiro定义了一个许可声明,定义了一个明确的行为或行动。 这是一个原始功能的声明在一个应用程序而已。 权限是最低级别构造安全策略,他们只明确定义应用程序可以做“什么”。

他们不描述"谁"能够执行的操作。

一些权限的例子:

- 打开一个文件
- 浏览'/user/list' 网页
- 打印文件
- 删除用户‘jsmith’

规定“谁”（用户）允许做“什么”（权限）在某种程度上是分配用权限的一种习惯做法。这始终是通过应用程序数据模型来完成的，并且在不同应用程序之间差异很大。

例如，权限可以组合到一个角色中，且该角色能够关联一个或多个用户对象。或者某些应用程序能够拥有一组用户，且这个组可以被分配一个角色，通过传递的关联，意味着所有在该组的用户隐式地获得了该角色的权限。

如何授予用户权限可以有很多变化——应用程序基于应用需求来决定如何使其模型化。

Wildcard Permissions 通配符的权限

上述权限的例子,“打开文件”、“浏览'/user/list' 网页”,等都是有效的权限。 然而,计算来解释这些自然语言字符串和确定用户是否允许执行这一行为这将是非常困难的。

为了使更容易处理但仍可读权限语句,Shiro 提供强大的和直观的语法我们称之为 WildcardPermission 。

Simple Usage 简单示例

你想保护访问贵公司的打印机,这样有些人可以打印到特定的打印机,而其他人可以查询什么工作目前在队列中。

一个非常简单的方法是使用授予用户“queryPrinter”权限。 然后你可以通过调用检查用户是否有 queryPrinter 权限:

```
subject.isPermitted("queryPrinter")
```

这是(大部分)相当于

```
subject.isPermitted( new WildcardPermission("queryPrinter") )
```

但远不只这些。

简单的权限字符串可能在简单的应用程序中工作的很好，但它需要你拥有像"printPrinter"，"queryPrinter"，"managePrinter"等权限。你还可以通过使用通配符授予用户"*"权限（赋予此权限构造它的名字），这意味着他们在整个应用程序中拥有了所有的权限。

但使用这种方法没有办法说明用户具有“所有打印机权限”。 出于这个原因,，Wildcard Permissions（通配符权限）支持多层次的权限管理。

Multiple Parts 多个部分

通配符权限支持多层次或部件（parts）的概念。例如，你可以通过授予用户权限来调整之前那个简单的例子

```
printer:query
```

在这个例子中的冒号是一个特殊字符，它用来分隔权限字符串的下一部件。

在该例中，第一部分是权限被操作的领域（printer），第二部分是被执行的操作（query）。上面其他的例子将被改为：

```
printer:print
printer:manage
```

对于能够使用的部件是没有数量限制的，因此它取决于你的想象，依据你可能在你的应用程序中使用的方法

Multiple Values

每个部件能够保护多个值。因此，除了授予用户 "printer:print" 和 "printer:query" 权限外，你可以简单地授予他们一个：

```
printer:print,query
```

它能够赋予用户 print 和query 打印机的能力。由于他们被授予了这两个操作，你可以通过调用下面的语句来判断用户是否有能力查询打印机：

```
subject.isPermitted("printer:query")
```

该语句将会返回true

All Values

如果你想在一个特定的部件给某一用户授予所有的值呢？这将是比手动列出每个值更为方便的事情。同样，基于通配符的话，我也可以做到这一点。若打印机域有3 个可能的操作（query，print 和manage），可以像下面这样：

```
printer:query,print,manage
```

简单点变成这样：

```
printer:*
```

然后，任何对 "printer:XXX" 的权限检查都将返回 true。以这种方式使用的通配符比明确地列出操作具有更好的尺度，如果你不久为应用程序增加了一个新的操作，你不需要更新使用通配符那部分的权限。

最后，在一个通配符权限字符串中的任何部分使用通配符 token 也是可以的。例如，如果你想对某个用户在所有领域（不仅仅是打印机）授"view"权限，你可以这样做：

```
*:view
```

这样任何对"foo:view"的权限检查都将返回true。

Instance-Level Access Control

另一种常见的通配符权限用法是塑造实例级的访问控制列表。在这种情况下，你使用三个部件——第一个是域，第二个是操作，第三个是被付诸实施的实例。

简单例子

```
printer:query:lp7200
printer:print:epsoncolor
```

第一个定义了查询拥有ID lp7200 的打印机的行为。第二条权限定义了打印到拥有 ID epsoncolor 的打印机的行为。

如果你授予这些权限给用户，那么他们能够在特定的实例上执行特定的行为。然后你可以在代码中做一个检查：

```
if ( SecurityUtils.getSubject().isPermitted("printer:query:lp7200") {
    // 返回ID lp7200 的打印机的当前任务
}
```

这是体现权限的一个极为有效的方法。但同样，为所有的打印机定义多个实例ID 能很好的扩展，尤其是当新的打印机添加到系统的时候。你可以使用通配符来代替：

```
printer:print:*
```

这个做到了扩展，因为它同时涵盖了任何新的打印机。你甚至可以运行访问所有打印机上的所有操作

```
printer:*:*
```

或在一台打印机上的所有操作：

```
printer:*:lp7200
```

或甚至特定的操作：

```
printer:query, print:lp7200
```

`*` 通配符， `,` 子部件分离器可用于权限的任何部分。

Missing Parts 缺省的部分

最后要注意的是权限分配：缺省的部分意味着用户可以访问所有与之匹配的值，换句话说

```
printer:print
```

等价于

```
printer:print:*
```

并且

```
printer
```

等价于

```
printer:*:*
```

然而，你只能从字符串的结尾处省略部件，因此这样的：

```
printer:lp7200
```

并不等价于

```
printer:*:lp7200
```

Checking Permissions 检查权限

虽然权限分配使用通配符较为方便且具有扩展性（"printer:print:*" = print to any printer），但在运行时的权限检查应该始终基于大多数具体的权限字符串。

例如，如果用户有一个用户界面，他们想打印一份文档到 lp7200 打印机，你应该通过执行这段代码来检查用户是否被允许这样做：

```
if ( SecurityUtils.getSubject().isPermitted("printer:print:lp7200") ) {
    //用 lp7200 printer 打印文档
}
```

这种检查非常具体和明确地反映了用户在那一时刻试图做的事情。 然而，下面这个运行是检查是不为理想的：

```
if ( SecurityUtils.getSubject().isPermitted("printer:print") ) {
    //打印文档
}
```

为什么？因为第二个例子表明“对于下面的代码块的执行，你必须能够打印到任何打印机”。但请记住 "pinter:print" 是等价 "priner:print:*" 的！

因此，这是一个不正确的检查。如果当前用户不具备打印到任何打印机的能力，仅仅只有打印到 lp7200 和 epsoncolor 的能力，该怎么办呢？那么上面的第二个例子也绝不允许他们打印到 lp7200 打印机，即使他们已被赋予了相应的能力！

因此，经验法则是在执行权限检查时，尽可能使用权限字符串。当然，上面的第二块可能是在应用程序中别处的一个有效检查，如果你真的想要执行该代码块，如果用户被允许打印到任何打印机（令人怀疑的，但有可能）。你的应用程序将决定检查哪些有意义，但一般情况下，越具体越好。

Implication, not Equality 蕴涵，不相等

为什么运行时权限检查应该尽可能的具体，但权限分配可以较为普通？这是因为权限检查是通过蕴含的逻辑来判断的——而不是通过相等检查。

也就是说，如果一个用户被分配了 user: 权限，这意味着该用户可以执行 *user:view* 操作。*"user:"* 字符串明显不等于 "user:view"，但前者包含了后者。"user:*" 描述了 "user:view" 所定义的功能的一个超集。

为了支持蕴含规则，所有的权限都被翻译到实现org.apache.shiro.authz.Permission 接口的的对象实例中。这是以便 蕴含逻辑能够在运行时执行，且蕴含逻辑通常比一个简单的字符串相等检查更为复杂。所有在本文档中描述的通配符行为实际上是由 org.apache.shiro.authz.permission.WildcardPermission 类实现的。下面是更多的一些通过蕴含逻辑访问的通配符权限字符串：

```
user:*
```

同时蕴含着删除一个用户的能力:

```
user:delete
```

同样地，

```
user:*:12345
```

同时蕴含着更新 ID 为12345 的用户帐户的能力：

```
user:update:12345
```

而且

```
printer
```

蕴含着打印到任何打印机的能力

```
printer:print
```

Performance Considerations

权限检查比简单的相等比较要复杂得多，因此运行时的蕴含逻辑必须执行每个分配的权限。当使用像上面展示的权限字符串时，你正在隐式地使用 Shiro 默认的 WildcardPermission，它能够执行必要的蕴含逻辑。

Shiro 对 Realm 实现的默认行为是，对于每一个权限验证（例如，调用 subject.isPermitted），所有分配给该用户的权限（在他们的组，角色中，或直接分配给他们）需要为蕴含逻辑进行单独的检查。Shiro 通过首次成功检查立即返回来“短路”该进程以提高性能，但它不是一颗银弹。

这通常是极快的，当用户，角色和权限缓存在内存中且使用了一个合适的CacheManager时，在Shiro 不支持的 Realm 实现中。只要知道使用此默认行为，当权限分配给用户或他们的角色或组增加时，执行检查的时间一定会增加。

如果一个 Realm 的实现者有一个更为高效的方式来检查权限并执行蕴含逻辑，尤其它如果是基于应用程序数据模型的，他们应该实现它作为 Realm isPermitted* 方法实现的一部分。默认的 Realm/WildcardPermission 存在的支持覆盖了大多数用例的80~90%，但它可能不是在运行时拥有大量权限需要存储或检查的应用程序的最佳解决方案。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

7. Realms

Realm 是可以访问程序特定的安全数据如用户、角色、权限等的一个组件。Realm 会将这些程序特定的安全数据转换成一种 Shiro 可以理解的形式，Shiro 就可以依次提供容易理解的 Subject 程序API而不管有多少数据源或者程序中你的数据如何组织。

Realm 通常和数据源是一对一的对应关系，如关系数据库，LDAP 目录，文件系统，或其他类似资源。因此，Realm 接口的实现使用数据源特定的API 来展示授权数据（角色，权限等），如JDBC，文件IO，Hibernate 或JPA，或其他数据访问API。

Realm 实质上就是一个特定安全的 DAO

因为这些数据源大多通常存储身份验证数据（如密码的凭证）以及授权数据（如角色或权限），每个 Shiro Realm 能够执行身份验证和授权操作。

Realm Configuration

如果使用 Shiro 的 ini 配置文件，你可以在[main]区域内像配置其它对象一样定义和引用 Realms，但是 Realm 在 secrityManager上的配置有两种方式：明确方式和隐含方式。

Explicit Assignment 明确指定(显式)

在迄今所知的INI配置文件的相关知识中，这是一种显示的配置方式。在定义一个或多个 Realm后，再将它们在securityManager上进行统一配置。

例如：

```
fooRealm = com.company.foo.Realm
barRealm = com.company.another.Realm
bazRealm = com.company.baz.Realm

securityManager.realms = $fooRealm, $barRealm, $bazRealm
```

明确设置是确定性的，你可以非常确切地知道哪个 realm 在使用并且知道它们执行的顺序。可以查看认证章节的 Authentication Sequence 了解 Realm 的执行顺序的影响效果

Implicit Assignment隐含方式(隐式)

Not Preferred(不推荐)

这种方法可能引发意想不到的行为，如果你改变 *realm* 定义的顺序的话。建议你避免使用此方法，并使用显式分配，它拥有确定的行为。该功能很可能在未来的 *Shiro* 版本中被废弃或移除。

如果出于某些原因你不想显式地配置 securityManager.realms 的属性，你可以允许 Shiro 检测所有配置好的 realm 并直接将它们指派给securityManager。

使用这种方法，realm 将会按照它们预先定义好的顺序来指派给 securityManager 实例。

也就是说，对于下面的 shiro.ini 示例：

```
blahRealm = com.company.blah.Realm
fooRealm = com.company.foo.Realm
barRealm = com.company.another.Realm

# no securityManager.realms assignment here
```

基本上和下面这一行具有相同的效果：

```
securityManager.realms = $blahRealm, $fooRealm, $barRealm
```

然而，实现隐式分配，只是 realm 定义的顺序直接影响到了它们在身份验证和授权尝试中的访问顺序。如果你改变它们定义的顺序，你将改变主要的认证章节的 Authentication Sequence 的Authentication Sequence 是如何起作用的。由于这个原因，以及保证明确的行为，我们推荐使用显式分配而不是隐式分配。

Realm Authentication

当你理解了Shiro 的主要 认证 工作流后，了解在一个授权尝试中当 Authenticator 与 Realm 交互时到底发生了什么是很重要的。

Supporting AuthenticationTokens

正如在认证流程中提到的，在一个 Realm 执行一个验证尝试之前，它的supports)方法被调用。只有在返回值为 true 的时候它的getAuthenticationInfo(token) 方法才会执行。

通常情况下，一个 realm 将检查提交的令牌类型（接口或类）确定自己是否可以处理它，例如，一个处理生物特性数据的Realm 可能一点也不理解 UsernamePasswordTokens，在这种情况下它将从支持函数中返回 false。

Handling supported AuthenticationTokens

如果一个Realm支持提交的验证令牌，验证将调用 Realm 的getAuthenticationInfo(token)) 方法，这是Realm 使用后台数据进行验证的一次有效尝试，顺序执行以下动作：

1.检查主要 principal (身份)令牌（用户身份信息）；

2.基于主要 principal (信息)，在数据源中查找对应的用户数据；

3.确定令牌支持的 credentials (凭证数据)和存储的数据相符；

4.如果凭证相符，返回一个AuthenticationInfo实例，里面封装了 Shiro 可以理解的用户数据。

5.如果证据不符，抛出 AuthenticationException异常。

这是所有Realm getAuthenticationInfo 实现的最高级别工作流，Realm 在这个过程中可以自由做自己想做的事情，比如记录日志，修改数据，以及其他，只要对于存储的数据和验证尝试来讲是合理的就行。

仅有一件事情是必须的，如果 credentials （凭证）和给定的 principal （主要信息）匹配，需要返回一个非空的 AuthenticationInfo 实例，用来表示来自数据源的 Subject 账户信息。

节约时间

直接实现 *Realm* 接口也许需要时间并容易出错，大部分用户选择继承 *AuthorizingRealm* 虚拟类，这个类实现了常用的认证和授权工作流，这会节省你的时间而且不易出错。

Credentials Matching 凭证匹配

在上述 realm 认证工作流中，一个 Realm 必须较验 Subject 提交的凭证（如密码）是否与存储在数据中的凭证相匹配，如果匹配，验证成功，系统保留已认证的终端用户身份。

*Realm*凭证匹配

检查提交的凭证是否与后台存储数据相匹配是每一个 *Realm* 的责任而不是 *Authenticator* 的责任，每一个 *Realm* 都具备与凭证形式及存储密切相关的技能，可以执行详细的凭证比对，而 *Authenticator* 只是一个普通的工作流组件。

凭证匹配的过程在所有程序中基本上是一样的，通常只是对比数据方式不同。要确保这个过程在必要时是可插拔和可定制的，AuthenticatingRealm 以及它的子类支持用 CredentialsMatcher 来执行一个凭证对比。

在找到用户数据之后，它和提交的 AuthenticationToken 一起传递给一个 CredentialsMatcher，后者用来检查提交的数据和存储的数据是否相匹配。

Shiro某些 CredentialsMatcher 实现可以使你开箱即用，比如 SimpleCredentialsMatcher 和 HashedCredentialsMatcher 实现，但如果你想配置一个自定义的实现来完成特定的对比逻辑，你可以这样做：

```
Realm myRealm = new com.company.shiro.realm.MyRealm();
CredentialsMatcher customMatcher = new com.company.shiro.realm.CustomCredentialsMatche
r();
myRealm.setCredentialsMatcher(customMatcher);
```

或者，使用 Shiro 的 INI配置文件

```
[main]
...
customMatcher = com.company.shiro.realm.CustomCredentialsMatcher
myRealm = com.company.shiro.realm.MyRealm
myRealm.credentialsMatcher = $customMatcher
...
```

Simple Equality Check 简单证明匹配

所有 Shiro 的开箱即用 Realm 默认使用一个 SimpleCredentialsMatcher，SimpleCredentialsMatcher 对存储的用户凭证和从 AuthenticationToken 提交的用户凭证直接执行相等检查。

例如，如果提交了一个UsernamePasswordToken，SimpleCredentialsMatcher 检查提交的密码与存储的密码是否完全相等。

SimpleCredentialsMatcher 不仅仅对字符串执行相同对比，它可以对大多数常用类型，如字符串、字符数组、字节数组、文件和输入流等执行对比，查看 JavaDoc 获取更多的信息。

Hashing Credentials 哈希凭证：

取代将凭证按它们原始形式存储并执行原始数据的对比，存储终端用户的凭证（如密码）更安全的办法是在存储数据之前，先进行 hash 运算。

这确保终端用户的凭证不会以他们原始的形式存储，没有人能知道其原始值。与明文原始比较相比这是一种更为安全的做法，有安全意识的程序会更喜欢这种方法。

要支持这种加密的 hash 策略，Shiro 为 Realm 配置提供了一个HashedCredentialsMatcher 实现替代之前的 SimpleCredentialsMatcher。

Hashing 凭证以及 hash 迭代的好处超出了该 Realm 文档的范围，可以在HashedCredentialsMatcher JavaDoc更详细地了解这些主要内容。

Hashing and Corresponding Matchers 哈希以及相符合的匹配

对于一个使用 Shiro 的程序，如何配置才能简单地做到这些？

Shiro 提供了多个 HashedCredentialsMatcher 子类实现，你必须在你的 Realm 上配置指定的实现来匹配你的凭证所使用的 hash 算法。

例发，假设你的程序使用用户名/密码对来进行验证，基于上述 hash 凭证的好处，你希望当创建用户时以 SHA-265 方式加密用户的密码，你可以加密用户输入的明文密码并保存加密值：

```
import org.apache.shiro.crypto.hash.Sha256Hash;
import org.apache.shiro.crypto.RandomNumberGenerator;
import org.apache.shiro.crypto.SecureRandomNumberGenerator;
...

//我们将使用一个随机数发生器产生的盐。
//这比使用一个用户名作为盐或不使用盐更加安全。
//Shiro使这个实现变得容易。
//
//注意一个正常的应用程序将引用属性而
//不是每次创建一个新的RNG：
RandomNumberGenerator rng = new SecureRandomNumberGenerator();
Object salt = rng.nextBytes();

//我们的纯文本密码经过散列随机盐和多次迭代，
//得到Base64编码的值（比Hex需要较少的空间）：
String hashedPasswordBase64 = new Sha256Hash(plainTextPassword, salt, 1024).toBase64()
;

User user = new User(username, hashedPasswordBase64);
//在新帐户保存盐。该 HashedCredentialsMatcher
//稍后再的登录尝试的时候会处理它：
user.setPasswordSalt(salt);
userDAO.create(user);
```

由于你使用 SHA-256 加密你的密码，你需要告诉 Shiro 使用相应的 HashedCredentialsMatcher 来检查你的 hashing 值，在这个例子中，我们为了加强安全创建了一个随机的 salt 并且执行 1024 Hash 迭代（查看HashedCredentialsMatcher JAVADoc了解为什么），下面的Shiro INI 配置来做这件工作。

```
[main]
...
credentialsMatcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
# base64 编码, 例子中没有 hex:
credentialsMatcher.storedCredentialsHexEncoded = false
credentialsMatcher.hashIterations = 1024
# 下面属性只在 Shiro 1.0 需要，在 1.1 及以后版本移除了:
credentialsMatcher.hashSalted = true

...
myRealm = com.company.....
myRealm.credentialsMatcher = $credentialsMatcher
...
```

HSaltedAuthenticationInfo

确保正常运行的最后一件要做的事情是你的Realm实现必须返回一个SaltedAuthenticationInfo 实例而不是普通的AuthenticationInfo，SaltedAuthenticationInfo 接口确保你在创建用户帐户时使用的salt（如上面调用的 user.setPasswordSalt(salt);）能被HashedCredentialsMatcher 引用。

HashedCredentialsMatcher 需要使用 salt 来对提交的 AuthenticationToken 执行相同的hashing 技术来对比提交的令牌是否与存储的数据相匹配，所以如果你对用户密码使用salting（你应该这么做），确保你的 Realm 实现在返回 SaltedAuthenticationInfo 实例时引用它。

Disabling Authentication 禁用

如果有理由，你不希望某个 Realm 对某个资源执行验证（或者因为你只想 Realm 去执行授权检查），你可以完全禁用 Realm 的认证支持，方法就是在 Realm 的 supports 方法中始终返回 false,这样，你的 Realm 将在整个验证过程中不再被使用。

当然如果你想验证 Subject，至少要配置一个支持 AuthenticationTokens 的 Realm。

Realm Authorization

待定。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

8. Session Management

Apache Shiro 提供安全框架界独一无二的东西：一个完整的企业级Session 解决方案，从最简单的命令行及智能手机应用到最大的集群企业Web 应用程序。

这对许多应用有着很大的影响——直到 Shiro 出现，如果你需要 session 支持，你需要部署你的应用程序到 Web 容器或使用EJB 有状态会话Bean。Shiro 的 Session 支持比这两种机制的使用和管理更为简单，而且它在适用于任何程序，不论容器。

即使你在一个 Servlet 或 EJB 容器中部署你的应用程序，仍然有令人信服的理由来使用 Shiro 的Session 支持而不是容器的。下面是一个 Shiro 的 Session 支持的最可取的功能列表：

特性

- **POJO/J2SE based(IoC friendly)** - Shiro 的一切（包括所有Session 和Session Management 方面）都是基于接口和 POJO 实现。这可以让你轻松地配置所有拥有任何 JavaBeans 兼容配置格式（如JSON，YAML，Spring XML 或类 似的机制）的会话组件。你也可以轻松地扩展 Shiro 的组件或编写你自己所需的来完全自定义 session management。
- **Easy Custom Session Storage** - 因为Shiro 的Session 对象是基于 POJO 的，会话数据可以很容易地存储在任意数量的数据源。这允许你自定义你的应用程序会话数据的确切位置——例如，文件系统，联网的分布式缓存，关系数据库，或专有的数据存储。
- **Container-Independent Clustering!** - Shiro 的会话可以很容易地聚集通过使用任何随手可用的网络缓存产品，像 Ehcache + Terracotta，Coherence，GigaSpaces，等等。这意味着你可以为Shiro 配置会话群集一次且仅一次，无论你部署到什么容器中，你的会话将以相同的方式聚集。不需要容器的具体配置！
- **Heterogeneous Client Access** - 与 EJB 或 web 会话不同，Shiro 会话可以被各种客户端技术“共享”。例如，一个桌面应用程序可以“看到”和“共享”同一个被使用的物理会话通过在 Web 应用程序中的同一用户。我们不知道除了 Shiro 以外的其他框架能够支持这一点。
- **Event Listeners** - 事件监听器允许你在会话生命周期监听生命周期事件。你可以侦听这些事件和对自定义应用程序的行为作出反应——例如，更新用户记录当他们的会话过期时。
- **Host Address Retention** - Shiro Sessions 从会话发起地方保留IP 地址或主机名。这允许你确定用户所在，并作出相应的反应（通常是在IP 分配确定的企业内部网络环境）。
- **Inactivity/Expiration Support** - 由于不活动导致会话过期如预期的那样，但它们可以延续很久通过 touch() 方法来保持它们“活着”，如果你希望的话。这在 RIA (富互联网应用) 环境非常有用，用户可能会使用桌面应用程序，但可能不会经常与服务器进行通信，但该服务器的会话不应过期。
- **Transparent Web Use** - Shiro 的网络支持，充分地实现和支持关于

Sessions（HttpSession 接口和它的所有相关的API）的 Servlet2.5 规范.这意味着你可以使用在现有 Web 应用程序中使用Shiro 会话，并且你不需要改变任何现有的 Web 代码。

- **Can be used for SSO** - 由于 Shiro 会话是基于POJO 的，它们可以很容易地存储在任何数据源，而且它们可以跨 程序“共享”如果需要的话。我们称之为"poor man's SSO"，并它可以用来提供简单的登录体验，由于共享的会话能够保留身份验证状态。

Using Sessions 使用

几乎与所有其他在Shiro 中的东西一样，你通过与当前执行的Subject 交互来获取Session：

```
Subject currentUser = SecurityUtils.getSubject();

Session session = currentUser.getSession();
session.setAttribute( "someKey", someValue);
```

subject.getSession() 方法是调用 currentUser.getSubject(true)的快捷方式。

对于那些熟悉 HttpServletRequest API 的，Subject.getSession(boolean create) 方法与 HttpServletRequest.getSession(boolean create) 方法有着异曲同工之效。

- 如果该Subject 已经拥有一个Session，则boolean 参数被忽略且Session 被立即返回。
- 如果该Subject 还没有一个Session 且create 参数为true，则创建一个新的会话并返回该会话。
- 如果该Subject 还没有一个Session 且create 参数为false，则不会创建新的会话且返回null。

Any Application 任何应用

getSession 要求能够在任何应用程序工作，甚至是非 *Web* 应用程序。

当开发框架代码来确保一个 Session 没有被创建是没有必要的时候，subject.getSession(false) 可以起到很好的作用。 当你获取了一个 Subject 的 Session 后，你可以用它来做许多事情，像设置或取得 attribute，设置其超时时间，以及 更多。请参见 Session 的 JavaDoc来了解一个单独的会话能够做什么。

The SessionManager

SessionManager，名如其意，在应用程序中为所有的 subject 管理Session —— 创建，删除，inactivity(失效)及验证，等等。如同其他在Shiro 中的核心结构组件一样，SessionManager 也是一个由 SecurityManager 维护的顶级组件。

默认的 SecurityManger 实现是默认使用开箱即用的DefaultSessionManager。DefaultSessionManager 的实现提供一个应用程序所需的所有企业级会话管理，如 Session 验证，orphan cleanup，等等。这可以在任何应用程序中使用。

Web Applications Web 应用程序使用不同*SessionManager* 实现。请参见 *Web* 文档获取*web-specific Session Management* 信息。

像其他被 SecurityManager 管理的组件一样，SessionManager 可以通过 JavaBean 风格的 getter/setter 方法在所有Shiro 默认 SecurityManager 实现（getSessionManager()/setSessionManager()）上获取或设置值。或者例如，如果在使用 shiro.ini 配置：

```
[main]
...
sessionManager = com.foo.my.SessionManagerImplementation
securityManager.sessionManager = $sessionManager
```

但从头开始创建一个 SessionManager 是一个复杂的任务且是大多数人不想亲自做的事情。Shiro 的开箱即用的SessionManager 实现是高度可定制的和可配置的，并满足大多数的需要。本文档的其余部分假定你将使用 Shiro 的默认 SessionManager 实现，当覆盖配置选项时。但请注意，你基本上可以创建或插入任何你想要的东西。

Session Timeout 超时

默认地，Shiro 的 SessionManager 实现默认是 30 分钟会话超时。也就是说，如果任何 Session 创建后闲置（未被使用，它的lastAccessedTime)未被更新）的时间超过了 30 分钟，那么该 Session 就被认为是过期的，且不允许再被使用。

你可以设置 SessionManager 默认实现的 globalSessionTimeout 属性来为所有的会话定义默认的超时时间。例如，如果你想超时时间是一个小时而不是 30 分钟：

```
[main]
...
# 3,600,000 milliseconds = 1 hour
securityManager.sessionManager.globalSessionTimeout = 3600000
```

Per-Session Timeout

上面的 globalSessionTimeout 值默认是为新建的 Session 使用的。你可以在每一个会话的基础上控制超时时间通过设置单独的会话 timeout)值。与上面的 globalSessionTimeout 一样，该值以毫秒（不是秒）为时间单位。

Session Listeners

Shiro 支持 SessionListener 概念来允许你对发生的重要会话作出反应。你可以实现 SessionListener 接口（或扩展易用的SessionListenerAdapter ）并与相应的会话操作作出反应。 由于默认的 SessionManager sessionListeners 属性是一个集合，你可以对 SessionManager 配置一个或多个 listener 实 现，就像其他在 shiro.ini 中的集合一样：

```
[main]
...
aSessionListener = com.foo.my.SessionListener
anotherSessionListener = com.foo.my.OtherSessionListener

securityManager.sessionManager.sessionListeners = $aSessionListener, $anotherSessionLi
stener, etc.
```

All Session Events 所有Session 事件

当任何会话发生事件时，*SessionListeners* 都会被通知——不仅仅是对一个特定的会话

Session Storage 存储

每当一个会话被创建或更新时，它的数据需要持久化到一个存储位置以便它能够被稍后的应用程序访问。同样地，当一个会话失效且不再被使用时，它需要从存储中删除以便会话数据存储空间不会被耗尽。SessionManager 实现委托这些 Create/Read/Update/Delete(CRUD) 操作为内部组件，同时，SessionDAO，反映了数据访问对象（DAO）设计模式。

SessionDAO 的权力是你能够实现该接口来与你想要的任何数据存储进行通信。这意味着你的会话数据可以驻留在内存中，文件系统，关系数据库或NoSQL 的数据存储，或其他任何你需要的位置。你得控制持久性行为。

你可以将任何 SessionDAO 实现作为一个属性配置在默认的SessionManager 实例上。例如，在shiro.ini 中：

```
[main]
...
sessionDAO = com.foo.my.SessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO
```

然而，正如你可能期望的那样，Shiro 已经有一些很好的SessionDAO 实现，你可以立即使用或实现你需要的子类。

Web Applications

上述的 *securityManager.sessionManager.sessionDAO = $sessionDAO* 作业仅在使用一个本地的 *Shiro* 会话管理器时才 工作。*Web* 应用程序默认不会使用本地的会话管理器，而是保持不支持*SessionDAO* 的 *Servlet Container* 的默认会话 管理器。如果你想基于 *Web* 应用程序

启用 *SessionDAO* 来自定义会话存储或会话群集，你将不得不首先配置一个本 地的*Web* 会话管理器。例如：

```
[main]
...
sessionManager = org.apache.shiro.web.session.mgt.DefaultWebSessionManager
securityManager.sessionManager = $sessionManager

# Configure a SessionDAO and then set it:
securityManager.sessionManager.sessionDAO = $sessionDAO
```

Configure a SessionDAO! 配置

Shiro 的默认配置本地 *SessionManagers* 使用仅内存 *Session* 存储。这是不适合大多数应用程序的。大多数生产应用程序想要配置提供的 *EHCache*（见下文）支持或提供自己的 *SessionDAO* 实现。

请注意 *Web* 应用程序默认使用基于 *servlet* 容器的 *SessionManager*，且没有这个问题。这也是使用 *Shiro* 本地 *SessionManager* 的唯一问题。

EHCache SessionDAO

EHCache 默认是没有启用的，但如果你不打算实现你自己的 SessionDAO，那么强烈地建议你为 Shiro 的 SessionManagerment 启用 EHCache 支持。EHCache SessionDAO 将会在内存中保存会话，并支持溢出到磁盘，若内存成为制约。这对生产程序确保你在运行时不会随机地“丢失”会话是非常好的。

Use EHCache as your default 设置 *EHCache* 为默认

如果你不准备编写一个自定义的 *SessionDAO*，则明确地在你的 *Shiro* 配置中启用 *EHCache*。*EHCache* 带来的好处远不止在 *Sessions*，缓存验证和授权数据方面。更多信息，请参见 *Caching* 文档。

Container-Independent Session Clustering 独立容器的*Session*聚类

*如果你急需独立的容器会话集群，EHCache 会是一个不错的选择。你可以显式地在 EHCache 之后插入TerraCotta，并拥有一个独立于容器集群的会话缓存。不必再担心 Tomcat，JBoss，Jetty，WebSphere 或WebLogic 特定的会话集群！

为会话启用 EHCache 是非常容易的。首先，确保在你的 classpath 中有shiro-ehcache-.jar 文件（请参见 Download 页面或使用 Maven 或 Ant+Ivy）。*

当在 classpath 中后，这第一个 shiro.ini 实例向你演示怎样为所有Shiro 的缓存需要（不只是会话支持）使用 EHCache：

```
[main]

sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
securityManager.sessionManager.sessionDAO = $sessionDAO

cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
securityManager.cacheManager = $cacheManager
```

最后一行，securityManager.cacheManager = $cacheManager，为所有 Shiro 的需要配置了一个 CacheManager。该CacheManager 实例会自动地直接传送到 SessionDAO（通过 EnterpriseCacheSessionDAO 实现 CacheManagerAware 接口的性质）。

然后，当 SessionManager 要求 EnterpriseCacheSessionDAO 去持久化一个 Session 时，它使用一个 EHCache 支持的 Cache 实现去存储Session 数据。

Web Applications

当使用 *Shiro* 本地的 *SessionManager* 实现时不要忘了分配*SessionDAO* 是一项功能。*Web* 应用程序默认使用基于容器的 *SessionManager*，它不支持 *SessionDAO*。如果你想在 *Web* 应用程序中使用基于 *EHCache* 的会话存储，配置一个 如上所述的 *Web SessionManager*。

EHCache Session Cache Configuration

默认地，EhCacheManager 使用一个 Shiro 特定的 ehcache.xml 文件来建立 Session 缓存区以及确保 Sessions 正常存取的必要设置。

然而，如果你想改变缓存设置，或想配置你自己的 ehcache.xml 或EHCache net.sf.ehcache.CacheManager 实例，你需要配置缓存区来确保Sessions 被正确地处理。

如果你查看默认的 ehcache.xml 文件，你会看到接下来的 shiro-activeSessionCache 缓存配置：

```
<cache name="shiro-activeSessionCache"
       maxElementsInMemory="10000"
       overflowToDisk="true"
       eternal="true"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
       diskPersistent="true"
       diskExpiryThreadIntervalSeconds="600"/>
```

如果你希望使用你自己的 ehcache.xml 文件，那么请确保你已经为 Shiro 所需的定义了一个类似的缓存项。很有可能 你会改变 maxElementsInMemory 的属性值来吻合你的需要。然而，至少下面两个存在于你自己配置中的属性是非常重要的：

- overflowToDisk="true" - 这确保当你溢出进程内存时，会话不丢失且能够被序列化到磁盘

上。

- eternal="true" - 确保缓存项（ Session 实例）永不过期或被缓存自动清除。这是很有必要的，因为 Shiro 基于计划过程完成自己的验证。如果我们关掉这项，缓存将会在 Shiro 不知道的情况下清扫这些 Sessions，这可能引起麻烦

EHCache Session Cache Name

默认地，EnterpriseCacheSessionDAO 向 CacheManager 寻求一个名为"shiro-activeSessionCache"的 Cache。该缓存的 name/region 将在 ehcache.xml 中配置，如上所述。

如果你想使用一个不同的名字而不是默认的，你可以在EnterpriseCacheSessionDAO 上配置名字，例如：

```
[main]
...
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO.activeSessionsCacheName = myname
...
```

只要确保在 ehcahe.xml 中有一项与名字匹配且你已经配置好了如上所述的 overflowToDisk="true" 和 eternal="true"。

Custom Session IDs

Shiro 的 SessionDAO 实现使用一个内置的 SessionIdGenerator 组件来产生一个新的 Session ID 当每次创建一个新的会话的时候。该 ID 生成后，被指派给新近创建的Session 实例，然后该Session 通过SessionDAO 被保存下来。

默认的SessionIdGenerator 是一个 JavaUuidSessionIdGenerator，它能产生基于Java UUIDs 的 String IDs。该实现能够支持所有的生产环境。

如果它不符合你的需要，你可以实现 SessionIdGenerator 接口并在Shiro 的 SessionDAO 实例上配置该实现。例如， 在 shiro.ini 中：

```
[main]
...
sessionIdGenerator = com.my.session.SessionIdGenerator
securityManager.sessionManager.sessionDAO.sessionIdGenerator = $sessionIdGenerator
```

Session Validation & Scheduling 验证和计划

Sessions 必须被验证，这样任何无效(过期或停止)的会话能够从会话数据存储中删除。这保证了数据存储不会由于不能再次使用的会话而导致写入超时。

由于性能上的原因，仅仅在Sessions 被访问（也就是subject.getSession()）时验证它们是否停止或过期。这意味着， 如果没有额外的定期验证，Session orphans(孤儿)将会开始填充会话数据存储。

一个常见的说明孤儿的例子是 Web 浏览器中的场景：比方说，用户登录到Web 应用程序并创建了一个会话来保留 数据（身份验证状态，购物车等）。如果用户不注销，并在应用程序不知道的情况下关闭了浏览器，则他们的会话 实质上是“躺在”会话数据存储的（孤儿）。SessionManager 没有办法检测用户不再使用他们的浏览器，同时该会话永远不会被再次访问（它是孤儿了）。

会话孤儿，如果它们没有定期清除，将会填充会话数据存储（这是很糟糕的）。因此，为了防止丢放孤儿，SessionManager 实现支持 SessionValidationScheduler 的概念。SessionValidationScheduler 负责定期地验证会话以确保 它们是否需要清理。

Default SessionValidationScheduler 默认

默认可用的 SessionValidationScheduler 在所有环境中都是ExecutorServiceSessionValidationScheduler，它使用 JDK ScheduledExecutorService 来控制验证频率。

默认地，该实现每小时执行一次验证。你可以通过指定一个新的 ExecutorServiceSessionValidationScheduler 实例并指 定不同的间隔（以毫秒为单位）改变速率来更改验证频率：

```
[main]
...
sessionValidationScheduler = org.apache.shiro.session.mgt.ExecutorServiceSessionValida
tionScheduler
# Default is 3,600,000 millis = 1 hour:
sessionValidationScheduler.interval = 3600000

securityManager.sessionManager.sessionValidationScheduler = $sessionValidationSchedule
r
```

Custom SessionValidationScheduler 自定义

如果你希望提供一个自定义的 SessionValidationScheduler 实现，你可以指定它作为默认的 SessionManager 实例的一个属性。例如，在shiro.ini 中：

```
[main]
…
sessionValidationScheduler = com.foo.my.SessionValidationScheduler
securityManager.sessionManager.sessionValidationScheduler = $sessionValidationSchedule
r
```

Disabling Session Validation 禁用

在某些情况下，你可能希望禁用会话验证项，由于你建立了一个超出了Shiro 控制的进程来为你执行验证。例如，也许你正在使用一个企业的 Cache 并依赖于缓存的Time To Live 设置来自动地去除旧的会话。或者也许你已经制定了一个计划任务来自动清理一个自定义的数据存储。在这些情况下你可以关掉 session validation scheduling：

```
[main]
...
securityManager.sessionManager.sessionValidationSchedulerEnabled = false
```

当会话从会话数据存储取回数据时它仍然会被验证，但这会禁用掉 Shiro 的定期验证。

Enable Session Validation somewhere

如果你关闭了 *Shiro* 的*session validation scheduler*，你必须通过其他的机制（计划任务等）来执行定期的会话验证。 这是保证会话孤儿不会填充数据存储的唯一方法。

Invalid Session Deletion 删除无效的Session

正如我们上面所说的，进行定期的会话验证主要目的是为了删除任何无效的（过期或停止）会话来确保它们不会占用会话数据存储。

默认地，某些应用程序可能不希望 Shiro 自动地删除会话。例如，如果一个应用程序已经提供了一个 SessionDAO 备份数据存储查询，也许是应用程序团队希望旧的或无效的会话在一定的时间内可用。这将允许团队对数据存储运行查询来判断，例如，在上周某个用户创建了多少个会话，或一个用户会话的持续时间，或与之类似报告类型的查询。

在这些情形中，你可以关闭 invalid session deletion 项。例如，在shiro.ini 中：

```
[main]
...
securityManager.sessionManager.deleteInvalidSessions = false
```

请注意！如果你关闭了它，你得为确保你的会话数据存储不耗尽它的空间。你必须自己从你的数据存储中删除无效的会话！

还要注意，即使你阻止了 Shiro 删除无效的会话，你仍然应该使用某种会话验证方式——要没通过 Shiro 的现有验证机制，要么通过一个你自己提供的自定义的机制（见上述的"Disabling Session Validation"获取更多）。验证机制将会更新你的会话记录以反映无效的状态（例如，什么时候它是无效的，它最后一次被访问是什么时候，等等），即使你在其他的一些时间将手动删除它们。

如果你配置 *Shiro* 来让它不会删除无效的会话，你得为确保你的会话数据存储不会耗尽它的空间负责。你必须亲自从你的数据存储删除无效的会话！ 另外请注意，禁用会话删除并不等同于禁用 *session validation schedule*（会话验证调度）。你应该总是使用一个会话验证调度机制——无论是 *Shiro* 直接支持或者是你自己的。

Session Clustering 会话集群

Apache Shiro 会话能力一个非常令人兴奋的事情是,你可以原生的集群 Subject 会话,不需要再担心你的容器环境。也就是说,如果您使用 Shiro 的原生会话并配置一个会话集群,可以,说,部署到 Jetty 和 Tomcat 开发环境,JBoss 或 Geronimo 的生产环境,或任何其他环境，不用担心容器/特定于环境的集群安装或配置。 Shiro 会话集群配置一次，无论您的部署环境如何，都能正常运行

因为 Shiro 的基于 pojo 的 n 层体系结构,使会话集群的集群机制非常简单,使会话持久性的水平。 也就是说,如果您配置集群 SessionDAO ,DAO 可以与集群交互机制, Shiro 的 SessionManager 不需要知道集群的问题。

Distributed Caches 分布式缓存

分布式缓存比如 Ehcache + TerraCotta , GigaSpaces Oracle Coherence , memcached (和许多其他)已经解决 distributed-data-at-the-persistence-level(分布式数据持久层) 问题。 因此在 Shiro 会话使用集群配置如同使用分布式缓存一样简单。

这使您的灵活性选择确切的集群机制,适用于你的环境。

缓存

请注意,当启用分布式/企业缓存在您的会话集群数据存储,下列两种情况之一必须是 *true* 的:

- 整个集群范围的分布式缓存有足够内存来保留所有的 活动/当前 会话
- 如果整个集群范围的分布式缓存没有足够的内存保留所有活动会话,它必须支持磁盘溢出,会话是不会丢失。

如果这两种情况都失败，会导致会话被随机丢失,对于终端用户来说这可能令人沮丧。

EnterpriseCacheSessionDAO

如您所料,Shiro 已经提供了 SessionDAO 的实现，将保存数据到 企业/分布式缓存。EnterpriseCacheSessionDAO 预计 Shiro 缓存 或 缓存管理器已经配置，所以它可以利用缓存机制。

例如,在 shiro.ini :

```
#This implementation would use your preferred distributed caching product's APIs:
activeSessionsCache = my.org.apache.shiro.cache.CacheImplementation

sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
sessionDAO.activeSessionsCache = $activeSessionsCache

securityManager.sessionManager.sessionDAO = $sessionDAO
```

虽然你可以将缓存实例直接注入到 SessionDAO 如上所示,它通常是更常见的配置一般是缓存管理器 使用 Shiro 的所有缓存的需求(会话以及身份验证和授权数据)。 在这种情况下,而不是直接配置 缓存实例,您会告诉 EnterpriseCacheSessionDAO 缓存管理器 中的缓存的名称，应该用于存储活动会话。 例如:

```
# This implementation would use your caching product's APIs:
cacheManager = my.org.apache.shiro.cache.CacheManagerImplementation

# Now configure the EnterpriseCacheSessionDAO and tell it what
# cache in the CacheManager should be used to store active sessions:
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
# This is the default value.  Change it if your CacheManager configured a different na
me:
sessionDAO.activeSessionsCacheName = shiro-activeSessionsCache
# Now have the native SessionManager use that DAO:
securityManager.sessionManager.sessionDAO = $sessionDAO

# Configure the above CacheManager on Shiro's SecurityManager
# to use it for all of Shiro's caching needs:
securityManager.cacheManager = $cacheManager
```

但是上面的配置有一些有点奇怪。 你注意到吗?

有趣的关于这个配置,在配置我们实际上告诉 SessionDAO 实例使用一个 缓存 或 缓存管理器 ! 所以如何 SessionDAO 使用分布式缓存吗?

当 Shiro 初始化 SecurityManager ,它将检查看看 SessionDAO 是否 实现了 CacheManagerAware 接口。 如果是这样,它将自动提供任何可用的全局配置 缓存管理器 。

所以,当 Shiro 评估 securityManager.cacheManager = $cacheManager 行,它就会发现 EnterpriseCacheSessionDAO 实现了 CacheManagerAware 接口和调用 setCacheManager 和你的配置方法 缓存管理器 作为方法的参数。

然后在运行时,当 EnterpriseCacheSessionDAO 需要 activeSessionsCache 它会问 缓存管理器 使用实例返回它 activeSessionsCacheName 作为查找得到的关键 缓存 实例。 那 缓存 实例(支持分布式缓存/企业产品的API)将用于存储和检索会话的所有 SessionDAO CRUD 操作。

Ehcache + Terracotta

这样一个分布式缓存解决方案,人们取得了成功在使用 Shiro 的Ehcache + Terracotta 配对。看到 Ehcache-hosted 分布式缓存与Terracotta 文档的全部细节和 Ehcache 如何启用分布式缓存。

一旦你得到 Terracotta 集群处理 Ehcache ,Shiro-specific 部分非常简单。 阅读并遵守 EHCache SessionDAO 文档,但是我们需要做出一些改变

Ehcache 会话缓存配置 引用之前 不工作 —— Terracotta 特定的配置是必要的。 下面是一个示例配置,测试正常工作。 其内容保存在一个文件并将其保存在一个 ehcache.xml 文件:

```
<ehcache>
    <terracottaConfig url="localhost:9510"/>
    <diskStore path="java.io.tmpdir/shiro-ehcache"/>
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="false"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120">
        <terracotta/>
   </defaultCache>
   <cache name="shiro-activeSessionCache"
       maxElementsInMemory="10000"
       eternal="true"
       timeToLiveSeconds="0"
       timeToIdleSeconds="0"
       diskPersistent="false"
       overflowToDisk="false"
       diskExpiryThreadIntervalSeconds="600">
       <terracotta/>
   </cache>
   <!-- Add more cache entries as desired, for example,
        Realm authc/authz caching: -->
</ehcache>
```

当然你想要改变你 e 条目引用适当的主机/端口 Terracotta 服务器阵列。 还要注意,与 以前的配置中, ehcache-activeSessionCache 元素不设置 diskPersistent 或 overflowToDisk 属性为 true 的。他们都应该是 假 作为真实值不支持在集群配置。

在你保存了 ehcache.xml 文件,我们需要在 Shiro 的配置中引用它。 假设你已经作了 terracotta 特定 ehcache.xml 文件在类路径的根,这是最后的 Shiro 配置,使 Terracotta + Ehcache 集群的 Shiro 的需要(包括会话):

```
sessionDAO = org.apache.shiro.session.mgt.eis.EnterpriseCacheSessionDAO
# This name matches a cache name in ehcache.xml:
sessionDAO.activeSessionsCacheName = shiro-activeSessionsCache
securityManager.sessionManager.sessionDAO = $sessionDAO

# Configure The EhCacheManager:
cacheManager = org.apache.shiro.cache.ehcache.EhCacheManager
cacheManager.cacheManagerConfigFile = classpath:ehcache.xml

# Configure the above CacheManager on Shiro's SecurityManager
# to use it for all of Shiro's caching needs:
securityManager.cacheManager = $cacheManager
```

请记住, 顺序很重要 。 通过配置 缓存管理器 在 SecurityManager 最后,我们确保可以传播到所有之前配置缓存管理器 CacheManagerAware 组件(如 EnterpriseCachingSessionDAO)。

Zookeeper

用户报告使用 Apache Zookeeper 来管理/协调分布式会话。 如果你有任何文档/评论关于这个工作,请提交到到 Shiro 邮件列表

Sessions and Subject State 状态

Stateful Applications (Sessions allowed) 有状态

默认地，Shiro 的SecurityManager 实现使用一个Subject 的Session 作为一种策略来为接下来的引用存储Subject 的身份 ID（PrincipalCollection）和验证状态（subject.isAuthenticated()）。这通常发生在一个Subject 登录后或当一个 Subject 的身份 ID 通过Remember 服务被发现后。

这个默认的方法有几个好处:

- 任何服务于请求，调用或消息的应用程序可以用请求/调用/消息的有效载荷关联会话ID，且这是Shiro 用入站 请求关联用户所有所必须的。例如，如果使用Subject.Builder，这是需要获取相关的Subject 所需的一切：

  Serializable sessionId = //get from the inbound request or remote method invocation payload Subject requestSubject = new Subject.Builder().sessionId(sessionId).buildSubject();

这给大多数Web 应用程序及任何编写远程处理或消息框架的人带来了令人难以置信的方便（这事实上是Shiro 的Web 支持在自己的框架代码内关联Subject 和ServletRequest）。

- 任何"RememberMe"身份基于一个能够在第一次访问就能持久化到会话的初始请求。这确保了Subject 被记住的身份可以跨请求保存而不需要反序列化及将它解释到每个请求。例如，在一个 Web 应用程序中，没有必要去读取每一个请求的加密RememberMe Cookie，如果该身份在会话中是已知的。这可是一个很好的性能提升。

Stateless Applications (Sessionless) 无状态

虽然上述的默认策略对于大多数应用程序而言是很好的（通常是可取的），但这对于尝试尽可能无状态的应用程序来说是不合适的。许多无状态的架构规定在请求中不能存在持久状态，这种情况下的 Sessions 不会被允许（一个会话其本质代表了持久状态）。

但这一要求带来一个便利的代价—— Subject 状态不能跨请求保留。这意味着有这一要求的应用程序必须确保 Subject 状态可以在每一个请求中以其他的方式代表。

这几乎总是通过验证每个由应用程序处理的请求/调用/消息来完成的。例如，大多数无状态 Web 应用程序通常支持这一点通过执行 HTTP 基本验证，允许浏览器验证每一个代表最终用户的请求

Disabling Subject State Session Storage 禁用 Subject 状态会话存储

在 Shiro 1.2 及以后开始，应用程序想禁用 Shiro 的内部实现策略——将Subject 状态持久化到会话，可以禁用所有 Subject 的这一项，通过下面的操作：

在shiro.ini 中，在 securityManager 上配置下面的属性：

```
[main]
...
securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false
...
```

这将防止 Shiro 使用 Subject 的会话来存储所有跨请求/调用/消息的Subject 状态。只要确保你对每个请求进行了身份验证，这样 Shiro 将会对给定的请求/调用/消息知道它的 Subject 是谁。

*Shiro*的需求 *vs.* 你的需求

使用 *Sessions* 作为存储策略将禁用 *Shiro* 本身的实现。它没有完全地禁用 *Sessions*。如果你的任何代码显式地调用 *subject.getSession()* 或 *subject.getSession(true)* ，一个*session* 仍然会被创建。

A Hybrid Approach 一个混合的方法

上面的shiro.ini 配置中的 (securityManager.subjectDAO.sessionStorageEvaluator.sessionStorageEnabled = false) 这一行将会禁用Shiro 为所有的Subject 使用Session 作为一种实现策略。

但，如果你想使用混合的方法呢？如果某些对象应该有会话而某些没有？这种混合法方法能够给许多应用程序带来好处。例如：

- 也许 human Subject（如 Web 浏览器用户）由于上面提供的好处能够使用Session。
- 也许non-human Subject（如 API 客户端或第三方应用程序）不应该创建session 由于它们与软件的交互可能会 间歇或不稳定。
- 也许所有某种确定类型的 Subject 或从某一确定位置访问系统的应该将状态保持在会话中，但所有其他的不应 该。 如果你需要这个混合方法，你可以实现一个 SessionStorageEvaluator。

SessionStorageEvaluator

在你想究竟控制哪个 Subject 能够在它们的 Session 中保存它们的状态的情况下，你可以实现

org.apache.shiro.mgt.SessionStorageEvaluator 接口，并告诉Shiro 哪个 Subject 支持会话存储。

该接口只有一个方法：

```
public interface SessionStorageEvaluator {

    public boolean isSessionStorageEnabled(Subject subject);

}
```

关于更详细的API 说明，请参见 SessionStorageEvaluator 的JavaDoc。 你可以实现这一接口，并检查 Subject，为了你可能做出这一决定的任何信息

Subject Inspection

但实现 isSessionStorageEnabled(subject)接口方法时，你可以一直查看 Subject 并访问任何你需要用来作出决定的东西。

当然所有期望的 Subject 方法都是可用的（gePrincipals()等），但特定环境的 Subject 实例也是有价值的。

例如，在 Web 应用程序中，如果该决定必须基于当前 ServletRequest 中的数据，你可以获取该 request 或该 response，因为运行时的Subjce 实例实际上就是一个 WebSubject 实例：

```
...
public boolean isSessionStorageEnabled(Subject subject) {
    boolean enabled = false;
    if (WebUtils.isWeb(Subject)) {
        HttpServletRequest request = WebUtils.getHttpRequest(subject);
        //set 'enabled' based on the current request.
    } else {
        //not a web request - maybe a RMI or daemon invocation?
        //set 'enabled' another way...
    }

    return enabled;
}
```

N.B.框架开发人员应该考虑到这种类型的访问，并确保任何请求/调用/消息上下文对象可用是同过特定环境下的 Subject 实现的。联系 Shiro 用户邮件列表，如果你想帮助设置它，为了你的框架/环境。

Configuration 配置

在你实现了 SessionStorageEvaluator 接口后，你可以在 shiro.ini 中配置它：

```
[main]
...
sessionStorageEvaluator = com.mycompany.shiro.subject.mgt.MySessionStorageEvaluator
securityManager.subjectDAO.sessionStorageEvaluator = $sessionStorageEvaluator

...
```

Web Applications

通常 Web 应用程序希望在每一个请求的基础上容易地启用或禁用会话的创建，不管是哪个 Subject 正在执行请求。这经常在支持 REST 及Messaging/RMI 构架上使用来产生很好的效果。例如，也许正常的终端用户（使用浏览器的人）被允许创建和使用会话，但远程的 API 客户端使用REST 或 SOAP，不该拥有会话（因为它们在每一个请求上验证， 常见于 REST/SOAP 体系结构）。

为了支持这种 hybrid/per-request （混合/每次请求）的能力，noSessionCreation 过滤器被添加到 Shiro 的默认"池"g过滤器中，为 Web 应用程序启用的。该过滤器将会阻止在请求期间创建新的会话来保证无状态的体验。在shiro.ini 的[urls]项中，你通常定义该过滤器在所有其它过滤器之前来确保会话永远不会被使用。

举例：

```
[urls]
...
/rest/** = noSessionCreation, authcBasic, ...
```

这个过滤器允许现有会话的任何会话操作，但不允许在过滤的请求创建新的会话。也就是说，在请求或没有会话存在的Subject 调用下面四个方法中的任何一个时，将会自动地触发一个 DisabledSessionException 异常：

- httpServletRequest.getSession()
- httpServletRequest.getSession(true)
- subject.getSession()
- subject.getSession(true)

如果一个 Subject 在访问 noSessionCreation-protected-URL（无会话创建保护的 URL）之前已经有一个会话，则上述的四种调用仍然会如预期工作。

最后，在所有情况下，下面的调用将始终被允许：

- httpServletRequest.getSession(false)
- subject.getSession(false)

9. Cryptography 密码

待定。TO DO. 可以参阅 Apache Shiro加密功能.

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

10. Web

Configuration 配置

将 Shiro 集成到任何 Web 应用程序的最简单的方法是在 web.xml 中配置 ContextListener 和 Filter，理解如何读取 Shiro 的 INI 配置文件。大部分的 INI 配置格式定义在 Configuration 页的 INI Sections 节，但我在这里我们将介绍一些额外的Web 的特定部分。

Using Spring?

Spring 框架用户将不执行此设置。如果你使用 *Spring*，你将要阅读关于*Spring* 特定的 *Web* 配置

Web.xml

Shiro 1.2 and later

在Shiro 1.2 及以后版本，标准的 Web 应用程序通过添加下面的 XML 块到 web.xml 来初始化 Shiro：

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class
>
</listener>

...

<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

这假设一个 Shiro INI Configuration 文件在以下两个位置任意一个，并使用最先发现的那个：

1. /WEB-INF/shiro.ini
2. 在classpath 根目录下shiro.ini 文件

下面是上述配置所做的事情：

- EnvironmentLoaderListener 初始化一个Shiro WebEnvironment 实例（其中包含 Shiro 需要的一切操作，包括 SecurityManager ），使得它在 ServletContext 中能够被访问。如果你需要在任何时候获得WebEnvironment 实例，你可以调用 WebUtils.getRequiredWebEnvironment（ServletContext）。
- ShiroFilter 将使用此 WebEnvironment 对任何过滤的请求执行所有必要的安全操作。
- 最后，filter-mapping 的定义确保了所有的请求被 ShiroFilter 过滤，建议大多数 Web 应用程序使用以确保任何请求是安全的。

ShiroFilter filter-mapping

它通常可取的做法是在任何其他 *filter-mapping* 声明之前定义 *ShiroFilter filter-mapping*，以确保 *Shiro* 也能在那些过滤器 下工作的很好。

Custom WebEnvironment Class

默认情况下，EnvironmentLoaderListener 将创建一个IniWebEnvironment 实例，呈现 Shiro 基于INI 文件的Configuration。如果你愿意，你可以在 web.xml 中指定一个自定义的 ServletContext context-param：

shiroEnvironmentClass</param-name> com.foo.bar.shiro.MyWebEnvironment</param-value> </context-param>

这允许你自定义一个如何解析和代表 WebEnvironment 实例的配置格式。你可以为自定义的行为对现有的 IniWebEnvironment 创建子类，或完全支持不同的配置格式。例如，如果有人想在XML 中配置Shiro 而不是在INI 中，他们可以创建一个基于XML 的实现，如 com.foo.bar.shiro.XmlWebEnviroment。

Custom Configuration Locations

IniWebEnvironment 将会去读取和加载 INI 配置文件。默认情况下，这个类会自动地在下面两个位置寻找 Shiro.ini 配置（按顺序）。

1. /WEB-INF/shiro.ini
2. classpath:shiro.ini

它将使用最先发现的那个。 然而，如果你想把你的配置放在另一位置，你可以在 web.xml 中用contex-param 指定该位置。

```
<context-param>
    <param-name>shiroConfigLocations</param-name>
    <param-value>YOUR_RESOURCE_LOCATION_HERE</param-value>
</context-param>
```

默认情况下，在 ServletContext.getResource 方法定义的规则下，param-value 是可以被解析的。例如， /WEB-INF/some/path/shiro.ini。

但你也可以指定具体的文件系统，如classpath 或URL 位置，通过使用Shiro 支持的合适的 ResourceUtils 类，例如：

- file://home/foobar/myapp/shiro.ini
- classpath:com/foo/bar/shiro.ini
- url:http://confighost.mycompany.com/myapp/shiro.ini

Shiro 1.1 and earlier

在Web 应用程序中使用Shiro 1.1 或更早版本的最简单的方法是定义IniShiroFilter 并指定一个 filter-mapping:

```
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
</filter>

...

<!-- Make sure any request you want accessible to Shiro is filtered. /* catches all --
>
<!-- requests.  Usually this filter mapping is defined first (before all others) to --
>
<!-- ensure that Shiro works in subsequent filters in the filter chain:             --
>
<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

该定义期望你的INI 配置是一个在 classpath 根目录的Shiro.ini 文件（如：classpath:shiro.ini）。

Custom Path

如果你不想将你的 INI 配置放在 /WEB-INF/shiro.ini 或classpath:shiro.ini，你可以指定一个自定义的资源位置，如果必 要的话。添加一个 configPath 的 init-param，并指定资源位置。

```
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
    <init-param>
        <param-name>configPath</param-name>
        <param-value>/WEB-INF/anotherFile.ini</param-value>
    </init-param>
</filter>

...
```

不合格的（不完整的组合或'non-prefixed'）configPath 值被假定为ServletContext 的资源路径，通过 ServletContext.getResource) 方法所定义的规则来解析。

ServletContext resource paths - Shiro 1.2+

ServletContext 资源路径是一个在 *Shiro 1.2* 即将推出的功能。在 *1.2* 被发布后，所有的 *configPath* 定义必须指定一个 *classpath:*，*file:*或*url:*前缀。

通过分别地使用 classpath:，url:，或file:前缀来指明classpath，url，或 filesystem 位置，你也可以指定其他非 ServletContext 资源位置。例如：

```
...
<init-param>
    <param-name>configPath</param-name>
    <param-value>url:http://configHost/myApp/shiro.ini</param-value>
</init-param>
...
```

Inline Config

最后，也可以将你的 INI 配置嵌入到 web.xml 中而不使用一个独立的 INI 文件。你可以通过使用 init-param 做到这点，而不是 configPath：

```
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.IniShiroFilter</filter-class>
    <init-param><param-name>config</param-name><param-value>

    # INI Config Here

    </param-value></init-param>
</filter>
...
```

内嵌配置对于小型的或简单的应用程序通常是很好用的，但是由于以下原因一般把它具体化到一个专用的 Shiro.ini 文件中：

- 你可能编辑了许多安全配置，不希望为 web.xml 添加版本控制。
- 你可能想从余下的 web.xml 配置中分离安全配置。
- 你的安全配置可能变得很大，你想保持 web.xml 的苗条并易于阅读。
- 你有个负责的编译系统，相同的 shiro 配置可能需要在多个地方被引用。

这取决于你——使用什么使你的项目更有意义。

Web INI configuration

除了在主要的 Configuration 章节描述的标准的[main]，[user]和[roles]项外，你可以在shiro.ini 文件中指定具有 web 特性的[urls]项：

```
# [main], [users] and [roles] above here
...
[urls]
...
```

[urls]项允许你做一些在我们已经见过的任何 Web 框架都不存在的东西：在你的应用程序中定义自适应过滤器链来匹配URL 路径！

这将更为灵活，功能更为强大，比你通常在 web.xml 中定义的过滤器链更为简洁：即使你从未使用任何 Shiro 提供的其他功能并仅仅使用了这个，但它即使是单独使用也是值得的。

[urls]

在urls 项的每一行格式如下：

URL_Ant_Path_Expression = Path_Specific_Filter_Chain

举例：

```
...
[urls]

/index.html = anon
/user/create = anon
/user/** = authc
/admin/** = authc, roles[administrator]
/rest/** = authc, rest
/remoting/rpc/** = authc, perms["remote:invoke"]
```

接下来我们将讨论这些行的具体含义。

URL Path Expressions

等号左边是一个与Web 应用程序上下文根目录相关的 Ant 风格的路径表达式。

例如，假设你有如下的[urls]行：

```
/account/** = ssl, authc
```

此行表明，“任何对我应用程序的/accout 或任何它的子路径（/account/foo, account/bar/baz，等等）的请求都将触发'ssl, authc'过滤器链”。我们将在下面讨论过滤器链。

请注意，所有的路径表达式都是相对于你的应用程序的上下文根目录而言的。这意味着如果某一天你在某个位置部署了你的应用程序，如 www.somehost.com/myapp ，然后又将它部署到了 www.anotherhost.com （没有'myapp'子目录），这样的匹配模式仍将继续工作。所有的路径都是相对于 HttpServletRequest.getContextPath()) 的值来的。

Order Matters! 秩序

URL 路径表达式按事先定义好的顺序判断传入的请求，并遵循 *FIRST MATCH WINS* 这一原则。例如，让我们假设有如下链的定义：

```
/account/** = ssl, authc
/account/signup = anon
```

如果传入的请求旨在访问 */account/signup/index.html*（所有 *'anon'ymous* 用户都能访问），那么它将永不会被处理！原因是因为*/account/*** 的模式第一个匹配了传入的请求，“短路”了其余的定义。 始终记住基于*FIRST MATCH WINS* 的原则定义你的过滤器链！

Filter Chain Definitions 定义

等号右边是逗号隔开的过滤器列表，用来执行匹配该路径的请求。它必须符合以下格式：

filter1[optional_config1], filter2[optional_config2], ..., filterN[optional_configN]

并且：

- filterN 是一个定义在[main]项中的 filter bean 的名字
- [optional_configN]是一个可选的括号内的对特定的路径，特定的过滤器有特定含义的字符串（每个过滤器，每个路径的具体配置！）。若果该过滤器对该 URL 路径并不需要特定的配置，你可以忽略括号，于是 filteNr[] 就变成了 filterN.

因为过滤器标志符定义了链（又名列表），所以请记住顺序问题！请按顺序定义好你的逗号分隔的列表，这样请求就能够流通这个链。

最后，每个过滤器按照它期望的方式自由的处理请求，即使不具备必要的条件（例如，执行一个重定向，响应一个 HTTP 错误代码，直接渲染等）。否则，它有可能允许该请求继续通过这个过滤器链到达最终的视图。

确保能够对路径的具体配置作出反应，即*[optional_configN]*一个过滤器标志符的一部分，是一个对所有*Shiro* 过滤器 适用的独有的功能。

你也可以这样做，如果你想创建你自己的 *javax.servlet.Filter* 的实现的话，确保你的过滤器子类 *org.apache.shiro.web.filter.PathMatchingFilter*

Available Filters 可用的过滤器

在过滤器链中能够使用的过滤器“池”被定义在[main]项。在[main]项中指派给它们的名字就是在过滤器链定义中使 用的名字。例如：

```
[main]
...
myFilter = com.company.web.some.FilterImplementation
myFilter.property1 = value1
...

[urls]
...
/some/path/** = myFilter
```

Default Filters

当运行一个 Web 应用程序时，Shiro 将会创建一些有用的默认 Filter 实例，并自动地在[main]项中将它们置为可用。

你可以在 main 中配置它们，当作在你的链的定义中你是否有任何其他的bean 和 reference。例如：

```
[main]
...
# Notice how we didn't define the class for the FormAuthenticationFilter ('authc') - i
t is instantiated and available already:
authc.loginUrl = /login.jsp
...

[urls]
...
# make sure the end-user is authenticated.  If not, redirect to the 'authc.loginUrl' a
bove,
# and after successful authentication, redirect them back to the original account page
 they
# were trying to view:
/account/** = authc
...
```

自动地可用的默认的Filter 实例是被 DefaultFilter 枚举定义的，枚举的名称字段是可供配置的名称。它们是：

Filter Name	Class
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
logout	org.apache.shiro.web.filter.authc.LogoutFilter
noSessionCreation	org.apache.shiro.web.filter.session.NoSessionCreationFilter
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authc.UserFilter

Enabling and Disabling Filters 启动、禁用过滤器

由于这是与任何过滤器链定义机制（web.xml，Shiro 的INI 等）相关的例子，你通过在过滤器链中包含它来启用过滤器，通过在过滤器链中移除它来禁用过滤器。

但在Shiro 1.2 中新增的一个新功能是不通过从过滤器链中移除过滤器来启用或禁用过滤器。如果启用（默认设置），那么请求将如预期一样过滤。如果禁用，那么该过滤器将允许请求立即通过到FilterChain 的下一个元素。你可以基于一般配置属性触发过滤器的启用状态，或者你甚至可以在每一个请求的基础上触发。

这是一个强大的概念，因为基于特定需求启用或禁用一个过滤器比更改静态过滤器链（这是永久的且固定的）定义更为方便。

Shiro 通过它的 OncePerRequestFilter 抽象父类来完成这点。所有 Shiro 的不受规范限制的过滤器实现子类实现这一点，因此不需要从过滤器链移除它们实现启用或禁用。如果你需要实现此功能，你可以为自己的过滤器实现继承这个类的子类。

SHIRO-224 将有望为任何过滤器使用这项功能，不仅仅只是那些OncePerRequestFilter 的子类。如果这对你很重要，请为这个issue 投票。

General Enabling/Disabling

OncePerRequestFilter（及其所有子类）支持 Enabling/Disabling 所有请求及 per-request 基础。 一般为所有的请求启用或禁用一个过滤器是通过设置其 enabled 属性为true 或 false。默认的设置是true 由于大多 数过滤器本质上是需要执行的，如果他们被配置在一个过滤器链中。 例如，在shiro.ini 中：

```
[main]
...
# configure Shiro's default 'ssl' filter to be disabled while testing:
ssl.enabled = false

[urls]
...
/some/path = ssl, authc
/another/path = ssl, roles[admin]
...
```

该例表明，许多潜在的URL 路径都需要请求必须通过SSL 连接保证。在开发中设置SSL 是令人沮丧且费时的。在开发时，你可以禁用ssl 过滤器。当部署产品时，你可以启用它通过一个配置属性——这比手动更改所有URL 路径或 维护两个Shiro 配置要容易得多。

Request-specific Enabling/Disabling

OncePerRequestFilter 实际上决定过滤器启用或禁用是基于它的isEnabled(request, response) 方法。该方法默认返回enabled 属性的值，该属性通常是用来enabling/disabling 上面提及的所有请求。如果你想启用或禁用一个基于特定标准的请求的过滤器，你可以通过覆盖OncePerRequestFilter 的 isEnabled(request, response) 方法来 执行更多特定的检查。

Path-specific Enabling/Disabling

Shiro 的 PathMatchingFilter（一个 OncePerRequestFilter 的子类）能够对基于被过滤的特定路径的配置作出反应。这 意味着你可以启用或禁用一个过滤器基于路径和特定路径配置，除了传入的request 和 response。 如果你需要能够对匹配的路径和特定路径配置作出反应来判断一个过滤器是否是启用的或禁用的，而不是通过覆盖 OncePerRequestFilter 的 isEnabled(request, reponse)方法，你应该是覆盖 PathMatchingFilter 的 isEnabled(request,response,path,pathConfig) 方法。

Session Management

Servlet Container Sessions

在 web 环境中,Shiro 的默认的会话管理器 SessionManager 实现是 ServletContainerSessionManager 。 这个非常简单的实现代表所有会话管理职责(包括会话集群如果 servlet 容器支持)运行 servlet 容器。 它本质上是一个桥 Shiro 会话 API 的 servlet 容器,没有别的。

使用这个默认的一个好处是,使用现有的 servlet 容器的应用程序会话配置(超时,任何特定容器集群机制等)将正常工作。

这个默认的缺点是,你与 servlet 容器的特定会话行为。 举个例子,如果你想集群会话,但你使用 Jetty 在生产 、测试和 Tomcat 容器配置(或代码)将不具有可移植性。

Servlet Container Session Timeout 容器会话超时

如果使用默认 servlet 容器支持,您配置将在您的web应用程序的会话超时 web . xml 文件。 例如:

```
<session-config>
  <!-- web.xml expects the session timeout in minutes: -->
  <session-timeout>30</session-timeout>
</session-config>
```

Native Sessions 本地会话

如果你想让你的会话配置设置和集群便携式在 servlet 容器(比如 Jetty 在测试中,但 Tomcat 或 JBoss 在生产环境中),或你想控制特定的会话/聚类特性,您可以启用 Shiro 的原生会话管理。

“Native”这个词在这里意味着 Shiro 的企业将被用来支持所有会话管理实现 Subject 和 httpservletrequest 会话和完全绕过 servlet容器。 但放心,Shiro 实现 Servlet 规范的相关部分直接所以任何现有的 web/http 相关代码符合预期和不需要“知道”,Shiro 透明地管理会话。

DefaultWebSessionManager

启用本地会话管理您的web应用程序,您需要配置一个本地 web-capable 会话管理器覆盖默认的基于 servlet 容器。 你可以通过配置的一个实例 DefaultWebSessionManager Shiro 的 SecurityManager 。 例如,在 shiro.ini :

```
[main]
...
sessionManager = org.apache.shiro.web.session.mgt.DefaultWebSessionManager
# configure properties (like session timeout) here if desired

# Use the configured native session manager:
securityManager.sessionManager = $sessionManager
```

一旦宣布,您可以配置 DefaultWebSessionManager 实例与本地会话选项会话超时和集群配置的描述 会话管理 部分。

Native Session Timeout本地会话超时

配置后 DefaultWebSessionManager 例如,会话超时配置中描述 会话管理:会话超时

Session Cookie

DefaultWebSessionManager 支持两种网络自身配置属性:

- sessionIdCookieEnabled (一个 boolean)
- sessionIdCookie, 一个 Cookie 实例.

Cookie as a template 作为模板

sessionIdCookie 属性本质上是一个模板,你配置 *Cookie* 实例属性,该模板将在运行时用一个适当的会话*ID*值设置在实际 *HTTP Cookie header*中

Session Cookie Configuration 配置

DefaultWebSessionManager 的 sessionIdCookie 默认实例是一个 SimpleCookie 。 这个简单的实现允许 JavaBeans-style 属性配置为所有你想要的相关属性配置在 http Cookie。

例如,您可以设置 Cookie 域

```
[main]
...
securityManager.sessionManager.sessionIdCookie.domain = foo.com
```

查看 SimpleCookie JavaDoc 额外的属性。

cookie 的默认名称 JSESSIONID 根据servlet规范。 此外,Shiro 的cookie 支持 HttpOnly 标识。 sessionIdCookie 设置 在默认情况下 httponly 为 true 为了额外的安全。

Shiro 的 Cookie 概念支持 HttpOnly 标识 甚至在 Servlet 2.4和 2.5 环境(而对原生的 Servlet API 只支持它在 2.6 或更高版本)。

Disabling the Session Cookie 禁用

如果您不希望使用会话 cookie,您可以禁用它们被配置使用 sessionIdCookieEnabled 属性为 false。 例如

```
[main]
...
securityManager.sessionManager.sessionIdCookieEnabled = false
```

Remember Me Services

Shiro 将执行 'rememberMe' 服务如果 AuthenticationToken 实现了 org.apache.shiro.authc.RememberMeAuthenticationToken 接口。该接口指定了一个方法

```
boolean isRememberMe();
```

如果该方法返回 true，Shiro 将会在整个会话中记住终端用户的身份ID

UsernamePasswordToken and RememberMe

经常使用的 *UsernamePasswordToken* 已经实现了*RememberMeAuthenticationToken* 接口，并支持 *rememberMe* 登录。

Programmatic Support 编程支持

要有计划性地使用 rememberMe ，你可以在一个支持该配置的类上把它的值设为 true。例如，使用标准的 UsernamePasswordToken：

```
UsernamePasswordToken token = new UsernamePasswordToken(username, password);

token.setRememberMe(true);

SecurityUtils.getSubject().login(token);
```

Form-based Login 基于表单的登录

对于 Web 应用程序而言，authc 过滤器默认是FormAuthenticationFilter 。它支持将 'rememberMe' 的布尔值作为一个 form/request 参数读取。默认地，它期望该 request 参数被命名为 rememberMe。下面是一个支持这点的 shiro.ini 配置的例子：

```
[main]
authc.loginUrl = /login.jsp

[urls]

# your login form page here:
login.jsp = authc
```

同时在你的 web form 中有一个名为 'rememberMe' 的checkbox。

```
<form ...>

   Username: <input type="text" name="username"/> <br/>
   Password: <input type="password" name="password"/>
   ...
   <input type="checkbox" name="rememberMe" value="true"/>Remember Me?
   ...
</form>
```

默认地，FormAuthenticationFilter 将会寻找名为 username，password 及 rememberMe 的 request 参数。如果这些不同于你使用的form 中的表单域名，你可能想在 FormAuthenticationFilter 上配置这些参数名。例如，在 shiro.ini 中：

```
[main]
...
authc.loginUrl = /whatever.jsp
authc.usernameParam = somethingOtherThanUsername
authc.passwordParam = somethingOtherThanPassword
authc.rememberMeParam = somethingOtherThanRememberMe
...
```

Cookie configuration

你可以通过设定 的各方面的 cookie 属性来配置 rememberMe cookie 是如何工作的。例如，在shiro.ini 中：

```
[main]
...

securityManager.rememberMeManager.cookie.name = foo
securityManager.rememberMeManager.cookie.maxAge = blah
...
```

请参见 CookieRememberMeManager 及 SimpleCookie 的 JavaDoc 支持来获取更多的配置属性

JSP / GSP Tag Library

Apache Shiro 提供了一个 Subject-aware JSP/GSP 标签库，它允许你控制你的 JSP，JSTL 或 GSP 页面基于当前 Subject 的状态进行输出。这对于根据身份个性化视图及当前用户所浏览的页面授权状态是相当有用的。

Tag Library Configuration

标签库描述文件 (TLD)被打包在 META-INF/shiro.tld 文件中的 shiro-web.jar 文件中。要使用任何标签，添加下面一行到你 JSP 页面（或任何你定义的页面指令）的顶部。

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
```

我们使用 shiro 前缀用以表明 shiro 标签库命名空间，当然你可以指定任何你喜欢的名字。

现在我们将讨论每一个标签，并展示它是如何用来渲染页面的。

The guest tag

guest 标签将显示它包含的内容，仅当当前的 Subject 被认为是 'guest' 时。'guest' 是指没有身份 ID 的任何 Subject。也 就是说，我们并不知道用户是谁，因为他们没有登录并且他们没有在上一次的访问中被记住（RememberMe 服务）。 例子：

```
<shiro:guest>
    Hi there!  Please <a href="login.jsp">Login</a> or <a href="signup.jsp">Signup</a>
 today!
</shiro:guest>
```

guest 标签与user 标签逻辑相反。

The user tag

user 标签将显示它包含的内容，仅当当前的 Subject 被认为是 'user' 时。'user' 在上下文中被定义为一个已知身份 ID 的 Subject，或是成功通过身份验证及通过'RememberMe'服务的。请注意这个标签在语义上与authenticated 标签是 不同的，authenticated 标签更为严格。

```
<shiro:user>
Welcome back John! Not John? Click <a href="login.jsp">here<a> to login.
</shiro:user>
```

usre 标签与 guest 标签逻辑相反。

The authenticated tag

仅仅只当当前用户在当前会话中成功地通过了身份验证 authenticated 标签才会显示包含的内容。它比 'user' 标签更为严格。它在逻辑上与'notAuthenticated'标签相反。

authenticated 标签只有当当前 Subject 在其当前的会话中成功地通过了身份验证才会显示包含的内容。它比 user 标签更为严格，authenticated 标签通常在敏感的工作流中用来确保身份 ID 是可靠的。

例子：

```
<shiro:authenticated>
    <a href="updateAccount.jsp">Update your contact information</a>.
</shiro:authenticated>
```

authenticated 标签与 notAuthenticated 标签逻辑相反。

The notAuthenticated tag

notAuthenticated 标签将会显示它所包含的内容，如果当前 Subject 还没有在其当前会话中成功地通过验证。

例子：

```
<shiro:notAuthenticated>
    Please <a href="login.jsp">login</a> in order to update your credit card informati
on.
</shiro:notAuthenticated>
```

notAuthenticated 标签与 Authenticated 标签逻辑相反。

The principal tag

principal 标签将会输出 Subject 的主体（标识属性）或主要的属性。 若没有任何标签属性，则标签将使用 principal 的 toString()值来呈现页面。例如（假设 principal 是一个字符串的用户名）：

```
Hello, <shiro:principal/>, how are you today?
```

这（大部分地）等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipal().toString() %>, how are you today?
```

Typed principal

principal 标签默认情况下，假定该 principal 输出的是subject.getPrincipal() 的值。但若你想输出一个不是主要 principal 的值，而是属于另一个 Subject 的 principal collection，你可以通过类型来获取该 principal 并输出该值。

例如，输出 Subject 的用户 ID（并不是 username ），假设该 ID 属于 principal collection：

```
User ID: <principal type="java.lang.Integer"/>
```

这（大部分地）等价于下面：

```
User ID: <%= SecurityUtils.getSubject().getPrincipals().oneByType(Integer.class).toStr
ing() %>
```

Principal property

但如果该 principal （是默认主要的 principal 或是上面的'typed' principal）是一个复杂的对象而不是一个简单的字符串，而且你希望引用该 principal 上的一个属性该怎么办呢？你可以使用 property 属性来来表示 property 的名称来理解（必须通过 JavaBeans 兼容的 getter 方法访问）。例如（假主要的principal 是一个User 对象）：

```
Hello, <shiro:principal property="firstName"/>, how are you today?
```

这（大部分地）等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipal().getFirstName().toString() %>, how
 are you today?
```

或者，结合类型属性：

```
Hello, <shiro:principal type="com.foo.User" property="firstName"/>, how are you today?
```

这也很大程度地等价于下面：

```
Hello, <%= SecurityUtils.getSubject().getPrincipals().oneByType(com.foo.User.class).ge
tFirstName().toString() %>, how are you today?
```

The hasRole tag

hasRole 标签将会显示它所包含的内容，仅当当前 Subject 被分配了具体的角色。

例如：

```
<shiro:hasRole name="administrator">
    <a href="admin.jsp">Administer the system</a>
</shiro:hasRole>
```

hasRole 标签与lacksRole 标签逻辑相反。

The lacksRole tag

lacksRole 标签将会显示它所包含的内容，仅当当前 Subject 未被分配具体的角色。

例如：

```
<shiro:lacksRole name="administrator">
Sorry, you are not allowed to administer the system.
</shiro:lacksRole>
```

The hasAnyRole tag

hasAnyRole 标签将会显示它所包含的内容，如果当前的 Subject 被分配了任意一个来自于逗号分隔的角色名列表中的具体角色。

例如：

```
<shiro:hasAnyRoles name="developer, project manager, administrator">
    You are either a developer, project manager, or administrator.
</shiro:lacksRole>
```

The hasPermission tag

hasPermission 标签将会显示它所包含的内容，仅当当前Subject“拥有”（蕴含）特定的权限。也就是说，用户具有特定的能力。

例如：

```
<shiro:hasPermission name="user:create">
    <a href="createUser.jsp">Create a new User</a>
</shiro:hasPermission>
```

hasPermission 标签与lacksPermission 标签逻辑相反。

The lacksPermission tag

lacksPermission 标签将会显示它所包含的内容，仅当当前Subject 没有拥有（蕴含）特定的权限。也就是说，用户没有特定的能力。

例如：

```
<shiro:lacksPermission name="user:delete">
    Sorry, you are not allowed to delete user accounts.
</shiro:hasPermission>
```

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

10.1. Configuration 配置

参见文档

10.2. urls (Path-based security) 基于路径的安全

参见文档

10.3. Default Filters 默认过滤器

参见文档

10.4. Session Management

参见文档

10.5. JSP Tag Library

参见文档

11. Caching 缓存

Shiro 开发团队明白在许多应用程序中性能是至关重要的。Caching 是从第一天开始第一个建立在 Shiro 中的一流功能，以确保安全操作保持尽可能的快。

然而，Caching 作为一个概念是 Shiro 的基本组成部分，实现一个完整的缓存机制是安全框架核心能力之外的事情。为此，Shiro 的缓存支持基本上是一个抽象的（包装）API，它将“坐”在一个基本的缓存机制产品（例如，Ehcache，OSCache，Terracotta，Coherence，GigaSpaces，JBossCache 等）之上。这允许Shiro 终端用户配置他们喜欢的任何缓存机制。

Caching API

Shiro 有三个重要的缓存接口：

- CacheManager - 负责所有缓存的主要管理组件，它返回 Cache 实例。
- Cache - 维护key/value 对。
- CacheManagerAware - 通过想要接收和使用 CacheManager 实例的组件来实现。

CacheManager 返回Cache 实例，各种不同的Shiro 组件使用这些Cache 实例来缓存必要的数据。任何实现了 CacheManagerAware 的 Shiro 组件将会自动地接收一个配置好的 CacheManager，该 CacheManager 能够用来获取 Cache 实例。

Shiro 的 SecurityManager 实现及所有 AuthorizingRealm 实现都实现了 CacheManagerAware 。如果你在 SecurityManager 上设置了 CacheManger，它反过来也会将它设置到实现了CacheManagerAware 的各种不同的 Realm 上（OO delegation）。例如，在 shiro.ini 中：

```
securityManager.realms = $myRealm1, $myRealm2, ..., $myRealmN
...
cacheManager = my.implementation.of.CacheManager
...
securityManager.cacheManager = $cacheManager
# at this point, the securityManager and all CacheManagerAware
# realms have been set with the cacheManager instance
```

CacheManager Implementations

Shiro提供了许多开箱即用 CacheManager 实现，你可能会发现有用的而不是执行你自己的。

MemoryConstrainedCacheManager

MemoryConstrainedCacheManager 是一个 缓存管理器 实现适合单个jvm 生产环境。 这不是集中/分布式,所以,如果你的应用程序跨越多个 JVM(例如 web 应用程序运行在多个 web 服务器),你想要缓存实体跨 JVM 访问,您将需要使用一个分布式缓存实现。

MemoryConstrainedCacheManager 管理 MapCache 情况下,一个 MapCache 每个命名的缓存实例。 每一个 MapCache Shiro 的实例支持 SoftHashMap 它可以自动调整大小本身基于应用程序的运行时内存约束/需求(通过利用 JDK 吗 softreference 实例)。

因为 MemoryConstrainedCacheManager 可以根据应用程序的内存配置文件自动调整大小本身,它的使用是安全的,在单个 jvm 的生产应用程序以及测试的需要。 然而,它没有更多的高级功能 suche 生存时间或 Time-to-Expire 设置缓存实体。 对于这些更先进的缓存管理功能,您可能会希望使用下面更先进的缓存管理器产品。

```
...
cacheManager = org.apache.shiro.cache.MemoryConstrainedCacheManager
...
securityManager.cacheManager = $cacheManager
```

HazelcastCacheManager

Shiro 1.3 or later

这个特性在当前版本中还没有发布。 *Shiro 1.3* 以后将会出现

待定。

EhCacheManager

待定。

Authorization Cache Invalidation 授权缓存失效

最后请注意, AuthorizingRealm 有一个 clearCachedAuthorizationInfo) 方法能够被子类调用，用来清除特殊账户缓存的授权信息。它通常被自定义逻辑调用，如果与之匹配的账户授权数据发生了改变（来确保下次的授权检查能够捕获新数据）。

12. Concurrency & Multithreading 并发与多线程

待定

13. Testing 测试

文档的这一部分介绍了在单元测试中如何使用Shiro。

What to know for tests

由于我们已经涉及到了 Subject reference，我们知道 Subject 是“当前执行”用户的特定安全视图，且该 Subject 实例绑定到一个线程来确保我们知道在线程执行期间的任何时间是谁在执行逻辑。

这意味着三个基本的东西必须始终出现，为了能够支持访问当前正在执行的Subject：

1. 必须创建一个 Subject 实例
2. Subject 实例必须绑定当前执行的线程。
3. 在线程完成执行后（或如果该线程执行抛出异常），该 Subject 必须解除绑定来确保该线程在任何线程池环境中保持'clean'。

Shiro 拥有为正在运行的应用程序自动地执行 绑定/解除绑定 逻辑的构建。例如，在 Web 应用程序中，当过滤一个请求时，Shiro 的根过滤器执行该逻辑。但由于测试环境和框架不同，我们需要自己选择自己的测试框架来执行此 绑定/解除绑定 逻辑。

Test Setup 设置

我们知道在创建一个 Subject 实例后，它必须被绑定线程。在该线程（或在这个例子中，是一个 test ）完成执行后，我们必须解除 Subject 的绑定来保持线程的 'clean'.

幸运的是，现代测试框架如 JUnit 和 TestNG 已经能够在本地支持'setup'和'teardown'的概念。我们可以利用这一支持来模拟 Shiro 在一个“完整的”应用程序中会做些什么。我们已经在下面创建了一个你能够在你自己的测试中使用的抽象基类——随意复制和修改如果你觉得合适的话。它能够在单元测试和集成测试中使用（我在本例中使用 JUnit， 但 TestNG 也能够工作得很好）：

AbstractShiroTest

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.UnavailableSecurityManagerException;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.subject.support.SubjectThreadState;
import org.apache.shiro.util.LifecycleUtils;
```

```
import org.apache.shiro.util.ThreadState;
import org.junit.AfterClass;

/**
 * Abstract test case enabling Shiro in test environments.
 */
public abstract class AbstractShiroTest {

    private static ThreadState subjectThreadState;

    public AbstractShiroTest() {
    }

    /**
     * Allows subclasses to set the currently executing {@link Subject} instance.
     *
     * @param subject the Subject instance
     */
    protected void setSubject(Subject subject) {
        clearSubject();
        subjectThreadState = createThreadState(subject);
        subjectThreadState.bind();
    }

    protected Subject getSubject() {
        return SecurityUtils.getSubject();
    }

    protected ThreadState createThreadState(Subject subject) {
        return new SubjectThreadState(subject);
    }

    /**
     * Clears Shiro's thread state, ensuring the thread remains clean for future test
execution.
     */
    protected void clearSubject() {
        doClearSubject();
    }

    private static void doClearSubject() {
        if (subjectThreadState != null) {
            subjectThreadState.clear();
            subjectThreadState = null;
        }
    }

    protected static void setSecurityManager(SecurityManager securityManager) {
        SecurityUtils.setSecurityManager(securityManager);
    }

    protected static SecurityManager getSecurityManager() {
        return SecurityUtils.getSecurityManager();
```

```
    }

    @AfterClass
    public static void tearDownShiro() {
        doClearSubject();
        try {
            SecurityManager securityManager = getSecurityManager();
            LifecycleUtils.destroy(securityManager);
        } catch (UnavailableSecurityManagerException e) {
            //we don't care about this when cleaning up the test environment
            //(for example, maybe the subclass is a unit test and it didn't
            // need a SecurityManager instance because it was using only
            // mock Subject instances)
        }
        setSecurityManager(null);
    }
}
```

Testing & Frameworks

在*AbstractShiroTest* 类中的代码使用 *Shiro* 的 *ThreadState* 概念及一个静态的*SecurityManager*。这些技术在测试和框架代码中是很有用的，但几乎不曾在应用程序代码中使用。

大多数使用 *Shiro* 工作的需要确保线程的一致性的终端用户，几乎总是使用 *Shiro* 的自动管理机制，即 *Subject.associateWith* 和*Subject.execute* 方法。这些方法包含在 *Subject thread association* 参考文献中。

Unit Testing

单元测试主要是测试你的代码，且你的代码是在有限的作用域内。当你考虑到 Shiro 时，你真正要关注的是你的代码能够与 Shiro 的API 正确的运行——你不会想做关于Shiro 的实现是否工作正常（这是 Shiro 开发团队在Shiro 的代码库必须确保的东西）的必要测试。

测试 Shiro 的实现是否与你的实现协同工作是真实的集成测试（下面讨论）。

ExampleShiroUnitTest

由于单元测试适用于测试你的逻辑（而不是你可能调用的任何实现），这对于模拟你逻辑所依赖的任何 API 来说是个很好的主意。这能够与 Shiro 工作得很好——你可以模拟 Subject 实例，并使它反映任何情况下你所需的反应，这些反应是处于测试的代码做出的。

但正如上文所述，在 Shiro 测试中关键是要记住在测试执行期间任何Subject 实例（模拟的或真实的）必须绑定到线程。因此，我们所需要做的是绑定模拟的 Subject 以确保如预期进行。

（这个例子使用 EasyMock，但 Mockito 也同样地工作得很好）：

```
import org.apache.shiro.subject.Subject;
import org.junit.After;
import org.junit.Test;

import static org.easymock.EasyMock.*;

/**
 * Simple example test class showing how one may perform unit tests for code that requ
ires Shiro APIs.
 */
public class ExampleShiroUnitTest extends AbstractShiroTest {

    @Test
    public void testSimple() {

        //1.  Create a mock authenticated Subject instance for the test to run:
        Subject subjectUnderTest = createNiceMock(Subject.class);
        expect(subjectUnderTest.isAuthenticated()).andReturn(true);

        //2. Bind the subject to the current thread:
        setSubject(subjectUnderTest);

        //perform test logic here.  Any call to
        //SecurityUtils.getSubject() directly (or nested in the
        //call stack) will work properly.
    }

    @After
    public void tearDownSubject() {
        //3. Unbind the subject from the current thread:
        clearSubject();
    }

}
```

正如你所看到的，我们没有设立一个 Shiro SecurityManager 实例或配置一个Realm 或任何像这样的东西。我们简单地创建一个模拟Subject 实例，并通过调用setSubject 方法将它绑定到线程。这将确保任何在我们测试代码中的调用或在代码中我们正测试的SecurityUtils.getSubject()正常工作。

请注意，setSubject 方法实现将绑定你的模拟 Subject 到线程，且它仍将存在，直到你通过一个不同的 Subject 调用 setSubject 或直到你明确地通过调用 clearSubject() 将它从线程中清除。 保持 Subject 绑定到该线程多长时间（或在一个不同的测试中用来交换一个新的实例）取决于你及你的测试需求。

tearDownSubject()

在实例中的 tearDownSubject() 方法使用了 Junit 4 的注释来确保该Subject 在每个测试方法执行后被清除，不管发生 什么。这要求你设立一个新的 Subject 实例并将它设置到每个需要执行的测试中。

然而这也不是绝对必要的。例如，你可以只每个测试开始时绑定一个新的Subject 实例（通过 setSubject），也就是说，使用 @Before-annotated 方法。但如果你将要这么做，你可以同时使用 @After tearDownSubject() 方法来保持对称及'clean'。

你可以手动地在每个方法中混合及匹配该 setup/teardown 逻辑或使用@Before 和 @After 注释只要你认为合适。所有测试完成后，AbstractShiroTest 超类在无论怎样都会将 Subject 从线程解除绑定，因为 @After 注释在它的 tearDownShiro() 方法中。

Integration Testing

现在我们讨论了单元测试的设置，让我们讨论一些关于集成测试的东西。集成测试是指测试跨API 边界的实现。例如，测试当调用B 实现时A 实现是否工作，且B 实现是否做它该做的事情。你同样也可以在Shiro 中轻松地执行集成测试。Shiro 的SecurityManager 实例及它所包含的东西（如 Realms 和 SessionManager 等）都是占用很少内存的非常轻量级的POJO。这意味着你可以为每一个你执行的测试类创建并销毁一个SecurityManager 实例。当你的集成测试运行时，它们将使用“真实的” SecurityManager，且与你应用程序中相像的 Subject 实例将会在运行时使用。

ExampleShiroIntegrationTest

下面的实例代码看起来与上面的单元测试实例几乎相同，但这3 个步骤却有些不同：

1. 现在有了step '0'，它用来设立一个“真实的” SecurityManager 实例。
2. Step 1 现在通过Subject.Builder 构造一个“真实的” Subject 实例，并将它绑定到线程。

线程的绑定与解除绑定（step 2 和 3 ）与单元测试实例中的作用一样。

```
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.subject.Subject;
import org.apache.shiro.util.Factory;
import org.junit.After;
import org.junit.BeforeClass;
import org.junit.Test;

public class ExampleShiroIntegrationTest extends AbstractShiroTest {

    @BeforeClass
    public static void beforeClass() {
        //0.  Build and set the SecurityManager used to build Subject instances used i
n your tests
        //    This typically only needs to be done once per class if your shiro.ini do
esn't change,
        //    otherwise, you'll need to do this logic in each test that is different
        Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:te
st.shiro.ini");
        setSecurityManager(factory.getInstance());
    }

    @Test
    public void testSimple() {
        //1.  Build the Subject instance for the test to run:
        Subject subjectUnderTest = new Subject.Builder(getSecurityManager()).buildSubj
ect();

        //2. Bind the subject to the current thread:
        setSubject(subjectUnderTest);

        //perform test logic here.  Any call to
        //SecurityUtils.getSubject() directly (or nested in the
        //call stack) will work properly.
    }

    @AfterClass
    public void tearDownSubject() {
        //3. Unbind the subject from the current thread:
        clearSubject();
    }
}
```

正如你所看到的，一个具体的 SecurityManager 实现被实例化，并通过setSecurityManager 方法使其余的测试能够对其进行访问。然后测试方法能够使用该 SecurityManager，当使用 Subject.Builder 后通过调用getSecurityManager() 方法。

还要注意 SecurityManager 实例在 @BeforeClass 设置方法中只被设置一次——一个对于大多数测试类较为普遍的做法。如果你想，你可以创建一个新的 SecurityManager 实例并在任何时候从任何测试方法通过setSerurityManager 来设置它——例如，你可能会引用两个不同

的.ini 文件来构建一个根据你的测试需求而来的新 SecurityManager。

最后，与单元测试例子一样，AbstractShiroTest 超类将会清除所有Shiro 产物（任何存在的 SecurityManager 及 Subject 实例）通过它的 @AfterClass tearDownShiro() 方法来确保该线程在下个测试类运行时是 'clean' 的。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

14. Custom Subjects 自定义 Subject

毫无疑问，在 Apache Shiro 中最重要的概念就是 Subject。 'Subject' 仅仅是一个安全术语，是指应用程序用户的特定安全的“视图”。一个 Shiro Subject 实例代表了一个单一应用程序用户的安全状态和操作。

这些操作包括：

- authentication(login)
- authorization(access control)
- session access
- logout

我们原本希望把它称为"User"由于这样“很有意义”，但是我们决定不这样做：太多的应用程序现有的 API 已经有自己的 User classes/frameworks，我们不希望和这些起冲突。此外，在安全领域，"Subject" 这一词实际上是公认的术语。

Shiro 的 API 为应用程序提供 Subject 为中心的编程范式支持。当编码应用程序逻辑时，大多数应用程序开发人员想知道谁才是当前正在执行的用户 。虽然应用程序通常能够通过它们自己的机制（ UserService 等）来查找任何用户，但涉及到安全性时，最重要的问题是“谁才是当前的用户？”。

虽然通过使用 SecurityManager 可以捕获任何 Subject，但只有基于当前 用户/Subject 的应用程序代码更自然，更直观。

The Currently Executing Subject 当前执行的 Subject

几乎在所有环境下，你能够获得当前执行的 Subject 通过使用

```
org.apache.shiro.SecurityUtils:Subject currentUser
```

getSubject() 方法调用一个独立的应用程序，该应用程序可以返回一个在应用程序特有位置上基于用户数据的Subject，在服务器环境中（如，Web 应用程序），它基于与当前线程或传入的请求相关的用户数据上获得 Subject 。

当你获得了当前的 Subject 后，你能够拿它做些什么？

如果你想在他们当前的 session 中使事情对用户变得可用，你可得的他们的 session:

```
Session session = currentUser.getSession();
session.setAttribute( "someKey", "aValue" );
```

Session 是一个 Shiro 的具体实例，它提供了大多数你经常要和HttpSessions 用到的东西，但有一些额外的好处和一个很大的区别：它不需要一个 HTTP 环境！

如果在 Web 应用程序内部部署，默认的 Session 将会是基于HttpSession 的。但是，在一个非 Web 环境中，像这个简单的 Quickstart，Shiro 将会默认自动地使用它的 Enterprise Session Management。这意味着你可以在你的应用程序中使用相同的 API，在任何层，无论部署环境。这打开了应用程序的全新世界，由于任何需要 session 的应用程序不再被强迫使用 HttpSession 或 EJB Stateful Session Beans。而且，任何客户端技术现在能够共享会话数据。

所以，你现在可以获取一个 Subject 以及他们的 Session。对于真正有用的东西像检查会怎么样呢，如果他们被允许做某些事——如对角色和权限的检查？

嗯，我只能对已知的用户做这些检查。我们的 Subject 实例代表了当前的用户，但谁又是实际上的当前用户呢？呃，他们都是匿名的——也就是说，直到他们至少登录一次。那么，让我们像下面这样做：

```
if ( !currentUser.isAuthenticated() ) {
    //collect user principals and credentials in a gui specific manner
    //such as username/password html form, X509 certificate, OpenID, etc.
    //We'll use the username/password example here since it is the most common.
    //(do you know what movie this is from? ;)
    UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");
    //this is all you have to do to support 'remember me' (no config - built in!):
    token.setRememberMe(true);
    currentUser.login(token);
}
```

那就是了！它再简单不过了。

但如果他们的登录尝试失败了会怎么样？你可以捕获各种各样的具体的异常来告诉你到底发生了什么：

```
try {
    currentUser.login( token );
    //if no exception, that's it, we're done!
} catch ( UnknownAccountException uae ) {
    //username wasn't in the system, show them an error message?
} catch ( IncorrectCredentialsException ice ) {
    //password didn't match, try again?
} catch ( LockedAccountException lae ) {
    //account for that username is locked - can't login.  Show them a message?
}
    ... more types exceptions to check if you want ...
} catch ( AuthenticationException ae ) {
    //unexpected condition - error?
}
```

你，作为 应用程序/GUI 开发人员，可以基于异常选择是否显示消息给终端用户（例如，"在系统中没有与该用户名对应的帐户。"）。有许多不同种类的异常供你检查，或者你可以抛出你自己自定义的异常，这些异常可能是Shiro 还未提供的。有关详情，请查看 AuthenticationException 的JavaDoc。

好了，现在，我们有了一个登录的用户，我们还有什么可以做的呢？

比方说，他们是谁：

```
//print their identifying principal (in this case, a username):
log.info( "User [" + currentUser.getPrincipal() + "] logged in successfully." );
```

我们还可以测试他们是否有特定的角色：

```
if ( currentUser.hasRole( "schwartz" ) ) {
    log.info("May the Schwartz be with you!" );
} else {
    log.info( "Hello, mere mortal." );
}
```

我们还能够判断他们是否有权限对一个确定类型的实体进行操作

```
if ( currentUser.isPermitted( "lightsaber:weild" ) ) {
    log.info("You may use a lightsaber ring.  Use it wisely.");
} else {
    log.info("Sorry, lightsaber rings are for schwartz masters only.");
}
```

此外，我们可以执行一个非常强大的实例级权限检查——它能够判断用户是否能够访问一个类型的具体实例：

```
if ( currentUser.isPermitted( "winnebago:drive:eagle5" ) ) {
    log.info("You are permitted to 'drive' the 'winnebago' with license plate (id) 'ea
gle5'.  " +
                "Here are the keys - have fun!");
} else {
    log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
}
```

小菜一碟，对吧？ 最后，当用户完成了对应用程序的使用时，他们可以注销：

```
currentUser.logout(); //removes all identifying information and invalidates their sess
ion too.
```

这个简单的API 包含了 90% 的 Shiro 终端用户在使用 Shiro 时将会处理的东西。

Custom Subject Instances 自定义 Subject 实例

Shiro 1.0 中添加了一个新特性，能够在特殊情况下构造 自定义/临时 的subject 实例。

只在特殊情况下使用*!*

你应该总是通过调用 *SubjectUtils.getSubject()* 来获得当前正在执行的 *Subject*； 创建自定义的 *Subject* 实例只应在特殊情况下进行。

当一些"特殊情况"是，这是可以很有用的：

- 系统启动/引导——当没有用户月系统交互时，代码应该作为一'system'或daemon 用户来执行。创建 Subject实例来代表一个特定的用户是值得的，这样引导代码能够以该用户（如admin）来执行。 鼓励这种做法是由于它能保证 utility/system 代码作为一个普通用户以同样的方式执行，以确保代码是一致的。 这使得代码更易于维护，因为你不必担心 system/daemon 方案的自定义代码块。
- 集成测试——你可能想创建 Subject 实例，在必要时可以在集成测试中使用。请参阅测试文档获取更多的内容。
- Daemon/background 进程的工作——当一个 daemon 或 background 进程执行时，它可能需要作为一个特定的用户来执行。

如果你已经有一个 *Subject* 的实例，并希望它提供给其他线程，你应该使用 `*Subject.associateWith*` 方法，而不是创建一个新的Subject 实例。*

好了，假设你仍然需要创建自定义的Subject 实例的情况下，让我们看看如何来做：

Subject.Builder

Subject.Builder 被制定得非常容易创建 Subject 实例，而无需知道构造细节。 Builder 最简单的用法是构造一个匿名的，session-less（无会话） Subject 的实例。

```
Subject subject = new Subject.Builder().buildSubject()
```

上面所展示的默认的Subject.Builder 无参构造函数将通过SecurityUtils.getSubject() 方法使用应用程序当前可访问的 SecurityManager 。你也可以指定被额外的构造函数使用的 SecurityManager 实例，如果你需要的话：

```
SecurityManager securityManager = //acquired from somewhere
Subject subject = new Subject.Builder(securityManager).buildSubject();
```

所有其他的 Subject.Builder 方法可以在 buildSubject()方法之前被调用，它们来提供关于如何构造 Subject 实例的上下文。例如，假如你拥有一个 session ID ，想取得"拥有"该 session 的 Subject（假设该 session 存在且未过期）：

```
Serializable sessionId = //acquired from somewhere
Subject subject = new Subject.Builder().sessionId(sessionId).buildSubject();
```

同样地，如你想创建一个Subject 实例来反映一个确定的身份：

```
Object userIdentity = //a long ID or String username, or whatever the "myRealm" requir
es
String realmName = "myRealm";
PrincipalCollection principals = new SimplePrincipalCollection(userIdentity, realmName
);
Subject subject = new Subject.Builder().principals(principals).buildSubject();
```

然后，你可以使用构造的 Subject 实例，如预期一样对它进行调用。但请注意：

构造的 Subject 实例不会由于应用程序（线程）的进一步使用而自动地绑定到应用程序（线程）。如果你想让它对于任何代码都能够方便地调用SecurityUtils.getSubject()，你必须确保创建好的 Subject 有一个线程与之关联。

Thread Association 线程关联

如上所述，只是构建一个 Subject 实例，并不与一个线程相关联——一个普通的必要条件是在线程执行期间任何对 SecurityUtils.getSubject() 的调用是否能正常工作。确保一个线程与一个 Subject 关联有三种途径：

- Automatic Association(自动关联)—— 通过 Sujbect.execute* 方法执行一个Callable 或 Runnable 方法会自动地绑定和解除绑定到线程的Subject，在Callable/Runnable 异常的

前后。

- Manual Association(手动关联)——你可以在当前执行的线程中手动地对Subject 实例进行绑定和解除绑定。这通常对框架开发人员非常有用。
- Different Thread(不同的线程)——通过调用Subject.associateWith* 方法将 Callable 或 Runnable 方法关联到 Subject，然后返回的 Callable/Runnable 方法在另一个线程中被执行。如果你需要为Subject 在另一个线程上执行工作的话，这是首选的方法。

了解线程关联最重要的是，两件事情必须始终发生：

1. Subject 绑定到线程，所以它在线程的所有执行点都是可用的。Shiro 做到这点通过它的 ThreadState 机制，该机制是在 ThreadLocal 上的一个抽象。
2. Subject 将在某点解除绑定，即使线程的执行结果是错误的。这将确保线程保持干净，并在pooled/reusable 线程环境中清除任何之前的Subject 状态。

这些原则保证在上述三个机制中发生。接下来阐述它们的用法。

Automatic Association 自动关联

如果你只需要一个 Subject 暂时与当前的线程相关联，同时你希望线程绑定和清理自动发生，Subject 的 Callable 或 Runnable 的直接执行正是你所需要的。在 Subject.execute 调用返回后，当前线程被保证当前状态与执行前的状态是一样的。这个机制是这三个中使用最广泛的。

例如，让我们假定你有一些逻辑在系统启动时需要执行。你希望作为一个特定用户执行代码块，但一旦逻辑完成后，你想确保 线程/环境 自动地恢复到正常。你可以通过调用 Subject.execute* 方法来做到：

```
Subject subject = //build or acquire subject
subject.execute( new Runnable() {
    public void run() {
        //subject is 'bound' to the current thread now
        //any SecurityUtils.getSubject() calls in any
        //code called from here will work
    }
});
//At this point, the Subject is no longer associated
//with the current thread and everything is as it was before
```

当然，Callable 的实例也能够被支持，所以你能够拥有返回值并捕获异常：

```
Subject subject = //build or acquire subject
MyResult result = subject.execute( new Callable<MyResult>() {
    public MyResult call() throws Exception {
        //subject is 'bound' to the current thread now
        //any SecurityUtils.getSubject() calls in any
        //code called from here will work
        ...
        //finish logic as this Subject
        ...
        return myResult;
    }
});
//At this point, the Subject is no longer associated
//with the current thread and everything is as it was before
```

这种方法在框架开发中也是很有用的。例如，Shiro 对 secure Spring remoting 的支持确保了远程调用能够作为一个特 定的 Subject 来执行：

```
Subject.Builder builder = new Subject.Builder();
//populate the builder's attributes based on the incoming RemoteInvocation
...
Subject subject = builder.buildSubject();

return subject.execute(new Callable() {
    public Object call() throws Exception {
        return invoke(invocation, targetObject);
    }
});
```

Manual Association 手动关联

虽然 Subject.execute* 方法能够在它们返回后自动地清理线程的状态，但有可能在一些情况下，你想自己管理 ThreadState。当结合 w/Shiro 时，这几乎总是在框架开发层次使用，但它很少在 bootstrap/daemon 情景下使用（上面 Subject.execute(callable) 例子使用得更为频繁）。

Guarantee Cleanup

关于这一机制最重要的是，你必须一直保证当前的线程在逻辑执行完后被清理，以确保在一个可重复使用或线程池的环境中没有一个线程状态腐化。

最好的做法是在try/finally 块保证清理：

```
Subject subject = new Subject.Builder()...
ThreadState threadState = new SubjectThreadState(subject);
threadState.bind();
try {
    //execute work as the built Subject
} finally {
    //ensure any state is cleaned so the thread won't be
    //corrupt in a reusable or pooled thread environment
    threadState.clear();
}
```

有趣的是，这正是 Subject.execute* 方法实际上所做的——它们只是在Callable 或 Runnable 执行前后自动地执行这个逻辑。Shiro 的 ShiroFilter 为 Web 应用程序执行几乎相同的逻辑（ShiroFilter 使用 Web 特定的 ThreadState 的实现，超出了本节的范围）

Web Use

不要在一个处理 *Web* 请求的进程中使用上述 *ThreadState* 代码示例。 *Web* 特定的 *ThreadState* 的实现使用 *Web* 请求代替。相反，确保*ShiroFilter* 拦截 *Web* 请求以确保 *Subject* 的 *building/binding/cleanup* 能够好好的完成。

A Different Thread

如果你有一个 Callable 或 Runnable 实例要以 Subject 来执行，你将自己执行 Callable 或 Runnable（或这将它移交给线程池或执行者或ExcutorService），你应该使用 Subject.associateWith* 方法。这些方法确保在最终执行的线程中保 留 Subject，且该 Subject 是可访问的。

Callable 例子：

```
Subject subject = new Subject.Builder()...
Callable work = //build/acquire a Callable instance.
//associate the work with the built subject so SecurityUtils.getSubject() calls works
properly:
work = subject.associateWith(work);
ExecutorService executorService = new java.util.concurrent.Executors.newCachedThreadPo
ol();
//execute the work on a different thread as the built Subject:
executor.execute(work);
```

Runnable 例子：

```
Subject subject = new Subject.Builder()...
Runnable work = //build/acquire a Runnable instance.
//associate the work with the built subject so SecurityUtils.getSubject() calls works
properly:
work = subject.associateWith(work);
Executor executor = new java.util.concurrent.Executors.newCachedThreadPool();
//execute the work on a different thread as the built Subject:
executor.execute(work);
```

Automatic Cleanup

associateWith 方法自动执行必要的线程清理，以取保现在在线程池环境中的*clean*。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

15. Spring Framework

本页涵盖了将 Shiro 集成到基于 Spring 的应用程序的方法。

Shiro 的 JavaBean 兼容性使得它非常适合通过 Spring XML 或其他基于 Spring 的配置机制。Shiro 应用程序需要一个具有单例SecurityManager 实例的应用程序。请注意，这不会是一个静态的单例，但应该只有一个应用程序能够使用的实例，无论它是否是静态单例的。

Standalone Applications

这里是在 Spring 应用程序中启用应用程序单例 SecurityManager 的最简单的方法：

```
<!-- 定义连接后台安全数据源的realm -->
<bean id="myRealm" class="...">
   ...
</bean>
<bean id="securityManager" class="org.apache.shiro.mgt.DefaultSecurityManager">
   <!-- 单realm应用。如果有多个realm，使用‘realms’属性代替 -->
   <property name="realm" ref="myRealm"/>
</bean>
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPost
Processor"/>
<!-- 最简单的集成，是将securityManager bean配置成一个静态单例，也就是让            SecurityUti
ls.*
下的所有方法在任何情况下都起作用。在web应用中不要将securityManager bean配置为静态单例，
具体方式请参阅下文中的‘Web Application’部分 -->
<bean class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
   <property name="staticMethod" value="org.apache.shiro.SecurityUtils.setSecurityMana
ger"/>
   <property name="arguments" ref="securityManager"/>
</bean>
```

Web Applications

Shiro 拥有对 Spring Web 应用程序的一流支持。在 Web 应用程序中，所有 Shiro 可访问的 web 请求必须通过一个主要的 Shiro 过滤器。该过滤器本身是极为强大的，允许临时的自定义过滤器链基于任何 URL 路径表达式执行。

在Shiro 1.0 之前，你不得不在 Spring web 应用程序中使用一个混合的方式，来定义 Shiro 过滤器及所有它在 web.xml 中的配置属性，但在Spring XML 中定义 SecurityManager。这有些令人沮丧，由于你不能把你的配置固定在一个地方，以及利用更为先进的 Spring 功能的配置能力，如 PropertyPlaceholderConfigurer 或抽象 bean 来固定通用配置。

现在在 Shiro 1.0 及以后版本中，所有 Shiro 配置都是在 Spring XML 中完成的，用来提供更为强健的 Spring 配置机制。

以下是如何在基于 Spring web 应用程序中配置 Shiro：

web.xml

除了其他 Spring web.xml 中的元素（ ContextLoaderListener，Log4jConfigListener 等等），定义下面的过滤器及过滤器映射：

```
<!-- filter-name对应applicationContext.xml中定义的名字为“shiroFilter”的bean -->
<filter>
    <filter-name>shiroFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    <init-param>
        <param-name>targetFilterLifecycle</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
...
<!-- 使用“/*”匹配所有请求，保证所有的可控请求都经过Shiro的过滤。通常这个filter-mapping
放置到最前面(其他filter-mapping前面)，保证它是过滤器链中第一个起作用的 -->
<filter-mapping>
    <filter-name>shiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

applicationContext.xml

在你的 applicationContext.xml 文件中，定义 web 支持的SecurityManager 和 'shiroFilter' bean 将会被 web.xml 引用。

```
<bean id="shiroFilter" class="org.apache.shiro.spring.web.ShiroFilterFactoryBean">
    <property name="securityManager" ref="securityManager"/>
    <!-- 可根据项目的URL进行替换 -->
    <property name="loginUrl" value="/login.jsp"/>
    <property name="successUrl" value="/home.jsp"/>
    <property name="unauthorizedUrl" value="/unauthorized.jsp"/>
    <!-- 因为每个已经定义的javax.servlet.Filter类型的bean都可以在链的定义中通过bean名
    称获取，所以filters属性不是必须出现的。但是可以根据需要通过filters属性替换filter
    实例或者为filter起别名 -->
    <property name="filters">
        <util:map>
            <entry key="anAlias" value-ref="someFilter"/>
        </util:map>
    </property>
    <property name="filterChainDefinitions">
        <value>
            # some example chain definitions:
            /admin/** = authc, roles[admin]
            /docs/** = authc, perms[document:read]
            /** = authc
            # more URL-to-FilterChain definitions here
        </value>
    </property>
</bean>
<!-- 定义应用上下文的 javax.servlet.Filter beans。这些beans 会被上面定义的shiroFilter自
动感知，并提供给“filterChainDefinitions”属性使用。或者也可根据需要手动的将他们添加在
shiroFilter bean的“filters”属性下的Map标签中。 -->
<bean id="someFilter" class="..."/>
<bean id="anotherFilter" class="..."> ... </bean>
...
<bean id="securityManager" class="org.apache.shiro.web.mgt.DefaultWebSecurityManager">
    <!-- 单realm应用。如果需要配置多个realm，使用“realms”属性 -->
    <property name="realm" ref="myRealm"/>
    <!-- 默认使用servlet容器session。下面是使用shiro 原生session的例子(细节请参考帮助文档)-->
    <!-- <property name="sessionMode" value="native"/> -->
</bean>
<bean id="lifecycleBeanPostProcessor" class="org.apache.shiro.spring.LifecycleBeanPost
Processor"/>
<!-- 定义连接后台安全数据源的realm -->
<bean id="myRealm" class="...">
...
</bean>
```

Enabling Shiro Annotations 启用注解

不论独立的应用程序还是 web 应用程序，都可以使用 Shiro 提供的注解进行安全检查。比如 @RequiresRoles, @RequiresPermissions 等。这些注解需要借助 Spring 的 AOP扫描使用 Shiro 注解的类，并在必要时进行安全逻辑验证。

下面让我们看下如何开启注解，其实很简单，只要在applicationContext.xml 中定义两个 bean 即可。

```
<!-- 开启Shiro注解的Spring配置方式的beans。在lifecycleBeanPostProcessor之后运行 -->
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreato
r" depends-on="lifecycleBeanPostProcessor"/>
<bean class="org.apache.shiro.spring.security.interceptor.AuthorizationAttributeSource
Advisor">
    <property name="securityManager" ref="securityManager"/>
</bean>
```

Secure Spring Remoting 远程安全

Shiro 的 Spring 远程支持有两部分组成：远程调用的客户端配置和接收、处理远程调用的服务器端配置。

Server-side Configuration 服务端配置

当 Shiro 的 Server 端接收到一个远程方法调用时，与远程调用相关的Subject 必须在接收线程执行时绑定到接收线程上，这项工作通过在applicationContext.xml 中定义 SecureRemoteInvocationExecutor bean 完成。

```
<!-- Spring远程安全确保每个远程方法调用都与一个负责安全验证的Subject绑定 -->
<!-- can be associated with a Subject for security checks. -->
<bean id="secureRemoteInvocationExecutor" class="org.apache.shiro.spring.remoting.Secu
reRemoteInvocationExecutor">
    <property name="securityManager" ref="securityManager"/>
</bean>
```

SecureRemoteInvocationExecutor 定义完成后，需要将它加入到Exporter 中，这个 Exporter 用于暴露向外提供的服务，而且 Exporter 的实现类由具体使用的远程处理机制和协议决定。定义 Exporter beans 请参照 Spring 的 Remoting 章节。

以基于 HTTP 的远程 SecureRemoteInvocationExecutor 为例。(remoteInvocationExecutor 属性引用自secureRemoteInvocationExecutor)

```
<bean name="/someService" class="org.springframework.remoting.httpinvoker.HttpInvokerS
erviceExporter">
    <property name="service" ref="someService"/>
    <property name="serviceInterface" value="com.pkg.service.SomeService"/>
    <property name="remoteInvocationExecutor" ref="secureRemoteInvocationExecutor"/>
</bean>
```

Client-side Configuration 客户端配置

当远程调用发生时，负责鉴别信息的Subject需要告知server远程方法是谁发起的 。如果客户端是基于Spring的，那么这种关联可以通过Shiro的SecureRemoteInvocationFactory 完成。

```
<bean id="secureRemoteInvocationFactory" class="org.apache.shiro.spring.remoting.Secur
eRemoteInvocationFactory"/>
```

然后将SecureRemoteInvocationFactory 添加到与协议相关的Spring远程ProxyFactoryBean 中。

以基于HTTP协议的远程ProxyFactoryBean为例。(remoteInvocationExecutor 属性引用自 secureRemoteInvocationExecutor)

```
<bean id="someService" class="org.springframework.remoting.httpinvoker.HttpInvokerProx
yFactoryBean">
    <property name="serviceUrl" value="http://host:port/remoting/someService"/>
    <property name="serviceInterface" value="com.pkg.service.SomeService"/>
    <property name="remoteInvocationFactory" ref="secureRemoteInvocationFactory"/>
</bean>
```

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

16. Guice

Shiro Guice 集成是在 Shiro 1.2 添加的。 这个页面覆盖了 Shiro 融入 基于 Guice 的应用程序使用标准方法 Guice 的约定和机制。 阅读这个集成文档之前,你应该至少有点熟悉 Guice。

Overview 概述

shiro-guice 提供了三个 Guice 模块可以包含在您的应用程序。

- ShiroModule
  - 提供基本的集成设置 SecurityManager ,任何 Realms 和任何其他配置 Shiro。
  - 使用这个模块通过扩展它并添加您自己的自定义配置。
- ShiroWebModule
  - 的延伸 ShiroModule 设置网络环境,还允许过滤器链配置。 这使用 Guice Servlet Module 配置过滤器,因此要求设置。
  - 就像 ShiroModule 使用这个模块,通过扩展它并添加您自己的自定义配置。
- ShiroAopModule
  - 使用 Guice AOP 实现Shiro AOP 注释。 这个模块主要是关心适应Shiro AnnotationMethodInterceptors 到 Guice 方法拦截器模型。
  - 通常使用这个模块通过简单地安装它。 然而,如果你有你自己的 Shiro AnnotationMethodInterceptors ,他们可以很容易地注册通过扩展它。

Getting Started 开始

最简单的配置是扩展 ShiroModule 安装您自己的 Realm 。

```
class MyShiroModule extends ShiroModule {
       protected void configureShiro() {
           try {
               bindRealm().toConstructor(IniRealm.class.getConstructor(Ini.class));
           } catch (NoSuchMethodException e) {
               addError(e);
           }
       }

       @Provides
       Ini loadShiroIni() {
           return Ini.fromResourcePath("classpath:shiro.ini");
       }
   }
```

在这种情况下,会在用户和角色配置 shiro.ini 文件。

Shiro.ini 使用 *Guice*

重要的是要注意,在上面的配置,只有 用户 和 角色 部分使用*ini*文件。

然后,Guice 模块是用来创建一个注射器,注射器是用来获得一个 SecurityManager 。 下面的例子具有同样目的的前三行 快速入门 的例子。

```
Injector injector = Guice.createInjector(new MyShiroModule());
SecurityManager securityManager = injector.getInstance(SecurityManager.class);
SecurityUtils.setSecurityManager(securityManager);
```

AOP

Shiro 包括几个注释和执行授权对于通过 AOP 方法拦截器非常有用。 它还提供了一个简单的 API 编写 Shiro-specific 方法拦截器。 shiro-guice 支持这个的 ShiroAopModule 。

要使用它,只需实例化并安装模块和应用程序模块和你 ShiroModule 。

```
Injector injector = Guice.createInjector(new MyShiroModule(), new ShiroAopModule(), ne
w MyApplicationModule());
```

如果你有写自定义拦截器,符合 Shiro 的 api,您可能会发现它有用的扩展 ShiroAopModule 。

```
class MyShiroAopModule extends ShiroAopModule {
        protected void configureInterceptors(AnnotationResolver resolver)
        {
            bindShiroInterceptor(new MyCustomAnnotationMethodInterceptor(resolver));
        }
```

Web

shiro-guice 的网络集成设计集成 Shiro 及其过滤范式与 Guice servlet 模块。 如果您正在使用 Shiro 在 web 环境中,并使用 Guice servlet 模块,那么你应该延长 ShiroWebModule 而不是 ShiroModule。 你的网络。 xml 应该设置 Guice servlet 模块的建议。

```
class MyShiroWebModule extends ShiroWebModule {
        MyShiroWebModule(ServletContext sc) {
            super(sc);
        }

        protected void configureShiroWeb() {
            try {
                bindRealm().toConstructor(IniRealm.class.getConstructor(Ini.class));
            } catch (NoSuchMethodException e) {
                addError(e);
            }

            addFilterChain("/public/**", ANON);
            addFilterChain("/stuff/allowed/**", AUTHC_BASIC, config(PERMS, "yes"));
            addFilterChain("/stuff/forbidden/**", AUTHC_BASIC, config(PERMS, "no"));
            addFilterChain("/**", AUTHC_BASIC);
        }

        @Provides
        Ini loadShiroIni() {
            return Ini.fromResourcePath("classpath:shiro.ini");
        }
    }
```

在前面的代码中,我们已经绑定的 IniRealm 和设置四个过滤器链。 这些链相当于以下 ini 配置。

```
[urls]
/public/** = anon
/stuff/allowed/** = authcBasic, perms["yes"]
/stuff/forbidden/** = authcBasic, perms["no"]
/** = authcBasic
```

shiro-guice,过滤器名称 Guice 的钥匙。 所有可用默认的 Shiro 过滤器是常量,但是你并不仅限于这些。 为了使用一个自定义过滤器过滤器链,你要做的事情

```
Key customFilter = Key.get(MyCustomFilter.class);

addFilterChain("/custom/**", customFilter);
```

我们仍然需要告诉 guice-servlets Shiro 过滤器。 自 ShiroWebModule 是私人的,guice-servlets 不给我们揭露一个过滤器映射的方法,我们必须手动绑定它。

```
ShiroWebModule.guiceFilterModule()
```

或者,在一个应用程序模块,

```
ShiroWebModule.bindGuiceFilter(binder())
```

Properties 属性

许多 Shiro 类暴露通过 setter 方法配置参数。 shiro-guice 注入这些如果找到一个绑定 @Named("shiro.{propName}")。 例如,设置会话超时,你可以做以下的事情。

```
bindConstant().annotatedWith(Names.named("shiro.globalSessionTimeout")).to(30000L);
```

如果这个范式对你不起作用,你也可以考虑使用提供者直接实例化对象和调用setter。

Injection of Shiro Objects 注入对象

shiro-guice 使用 Guice TypeListener 对本地执行注射 Shiro 类(类的子目录 org.apache.shiro 但不是 org.apache.shiro.guice)。 然而, Guice 只考虑显式绑定类型作为候选人 TypeListeners ,所以如果你有一 个Shiro 对象,你想要注射,你必须显式声明它。 例如,设置 credentialsmatcher 领域,我们需要添加以下绑定:

```
  bind(CredentialsMatcher.class).to(HashedCredentialsMatcher.class);
  bind(HashedCredentialsMatcher.class);
  bindConstant().annotatedWith(Names.named("shiro.hashAlgorithmName")).to(Md5Hash.ALGO
RITHM_NAME);
```

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

17. CAS

shiro-cas 模块是用来保护一个 Jasig CAS 单点登录服务器。它使一个 Shiro-enabled 程序变成 CAS 客户端

Basic understanding of the CAS protocol CAS 协议的基本理解

1. 如果你想访问一个应用程序由 CAS 保护，并且如果你不验证在这个应用程序中的客户端，你重定向通过 CAS 客户端 到 CAS 服务器登录页面。 在 CAS 登录 url 定义了应用程序用户希望登录服务参数。

   http://application.examples.com/protected/index.jsp → HTTP 302 → https://server.cas.com/login?service=http://application.examples.com/shiro-cas

2. 你填写的登录名和密码和验证CAS服务器，然后将用户重定向到应用程序(服务的 url)和 url 的服务票证。 服务票证是短暂的一次性令牌可赎回在CAS服务器用户标识符(和可选地,用户属性)。

   https://server.cas.com/login?service=http://application.examples.com/shiro-cas → HTTP 302 → http://application.examples.com/shiro-cas?ticket=ST-4545454542121-cas

3. 应用程序直接问 CAS 服务器如果服务票是有效和 CAS 服务器响应通过身份验证的用户的身份。 一般来说,CAS 端转发用户最初叫保护页面。

   http://application.examples.com/shiro-cas?ticket=ST-4545454542121-cas → HTTP 302 → http://application.examples.com/protected/index.jsp

How to configure shiro to work with CAS server？配置

Dependency 依赖

```
<dependency>
    <groupId>org.apache.shiro</groupId>
    <artifactId>shiro-cas</artifactId>
    <version>version</version>
</dependency>
```

(版本 > = 1.2.0)。

CasFilter

你必须定义服务应用程序的 ur l(在 CAS 服务器也必须声明)。 该url将被用来接收 CAS 服务票证。 例如: http://application.examples.com/shiro-cas

在 shiro 配置,您必须定义 CasFilter :

```
[main]
casFilter = org.apache.shiro.cas.CasFilter
casFilter.failureUrl = /error.jsp
```

(失败的 url 时,服务票验证失败)。

和url可用:

```
[urls]
/shiro-cas = casFilter
```

这样,当用户通过 CAS服务器使用有效的服务票证(身份验证)后， 被重定向到应用程序服务 url (/shiro-cas),这个过滤器接收服务票证,并创建一个 CasToken 可以被 CasRealm 使用 。

CasRealm

CasRealm 使用 CasFilter 创建的 CasToken 通过 CAS 对CAS服务器服务票证 查验，从而对用户进行身份验证

在shiro配置,您必须添加 CasRealm :

```
[main]
casRealm = org.apache.shiro.cas.CasRealm
casRealm.defaultRoles = ROLE_USER
#casRealm.defaultPermissions
#casRealm.roleAttributeNames
#casRealm.permissionAttributeNames
#casRealm.validationProtocol = SAML
casRealm.casServerUrlPrefix = https://server.cas.com/
casRealm.casService = http://application.examples.com/shiro-cas
```

casServerUrlPrefix 是 CAS 服务器的 url(例如: https://server.cas.com)。

casService 是应用程序服务 url,url 指向应用程序收到 CAS 服务票证(例如: http://application.examples.com/shiro-cas)。

validationProcol可以是 SAML或 CAS(默认):属性 和 remember me信息 只推到 SAML 验证协议(具体定制除外)。 这取决于 CAS 服务器的版本:SAML 协议可以使用 CAS 服务器版本 >= 3.1。

如果选择 *SAML* 验证，需要添加更多依赖

```
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
</dependency>
<dependency>
    <groupId>org.opensaml</groupId>
    <artifactId>opensaml</artifactId>
    <version>1.1</version>
</dependency>
<dependency>
    <groupId>org.apache.santuario</groupId>
    <artifactId>xmlsec</artifactId>
    <version>1.4.3</version>
</dependency>
```

defaultRoles 是默认的角色给了 CAS 认证成功后通过身份验证的用户。

defaultPermissions 是默认的权限给了 CAS 认证成功后通过身份验证的用户。

roleAttributeNames 定义属性的名称来自 CAS 响应定义角色给了身份验证的用户(角色由 comas 进行分隔)。

permissionAttributeNames 定义属性的名称来自 CAS 响应它定义权限给身份验证的用户(权限由 comas 进行分隔)。

CasSubjectFactory

在 CAS 服务器,你可以“记住我”的支持。 这些信息是通过 SAML 验证或CAS 定制的验证。 反映在 Shiro CAS-remember 我地位,你必须定义一个特定的 CasSubjectFactory 在你的Shiro配置:

```
[main]
casSubjectFactory = org.apache.shiro.cas.CasSubjectFactory
securityManager.subjectFactory = $casSubjectFactory
```

Security of the application应用程序的安全

最后,您必须定义您的应用程序的安全。

在 Shiro 配置,您必须保护 url 与角色(例如):

```
[urls]
/protected/** = roles[ROLE_USER]
/** = anon
```

和登录 url 如果用户没有经过身份验证的 CAS 服务器上定义与应用程序服务网址:

```
[main]
roles.loginUrl = https://server.cas.com/login?service=http://application.examples.com/
shiro-cas
```

这样,如果你不验证和尝试访问 /保护/ url,您被重定向到CAS服务器进行身份验证。

Complete configuration sample 完整的配置样例

```
[main]
casFilter = org.apache.shiro.cas.CasFilter
casFilter.failureUrl = /error.jsp

casRealm = org.apache.shiro.cas.CasRealm
casRealm.defaultRoles = ROLE_USER
casRealm.casServerUrlPrefix = https://server.cas.com/
casRealm.casService = http://application.examples.com/shiro-cas

casSubjectFactory = org.apache.shiro.cas.CasSubjectFactory
securityManager.subjectFactory = $casSubjectFactory

roles.loginUrl = https://server.cas.com/login?service=http://application.examples.com/
shiro-cas

[urls]
/shiro-cas = casFilter
/protected/** = roles[ROLE_USER]
/** = anon
```

History 历史

Version 1.2.0 : shiro-cas 模块第一个发布版本.

18. Command Line Hasher

Shiro 1.2.0 及以后版本提供了一个命令行程序,可以哈希字符串和资源(文件、url、classpath 、 实体)几乎任何类型。 要使用它,您必须安装一个 Java 虚拟机，并且“Java”命令必须能访问访问 $PATH 环境变量。

Usage 使用

确保你可以访问shiro-tools-hasher-version-cli.jar 文件。 你可以发现这在 buildroot/tools/hasher/target 目录的源码构建或通过Maven下载。

一旦你获得 jar,您可以运行下面的命令:

> java -jar shiro-tools-hasher-X.X.X-cli.jar

这将打印所有可用选项标准(MD5、SHA1)和更复杂的密码散列的场景。

Common Scenarios 常见的场景

请参阅上面的命令打印指令。 它将提供一个详尽的清单的指令将帮助您根据您的需要使用厨师。 然而,我们已经提供了一些快速参考用途/场景下面方便。

shiro.ini 用户密码

最好是保持用户的密码 shiro.ini (用户) 部分安全。 添加一个新用户帐户,使用上面的命令 p (或 ——密码)选项:

> java -jar shiro-tools-hasher-X.X.X-cli.jar -p

它会问你输入密码确认

```
Password to hash:
Password to hash (confirm):
```

当命令执行，将会输出 securely-salted-iterated-and-hashed 密码，举例：

```
$shiro1$SHA-256$500000$eWpVX2tGX7WCP2J+jMCNqw==$it/NRclMOHrfOvhAEFZ0mxIZRdbcfqIBdwdwdD
XW2dM=
```

把这个值,并将其作为密码的用户定义行(其次是任何可选角色)中定义的 INI 用户配置 文档。例如

```
[users]
...
user1 = $shiro1$SHA-256$500000$eWpVX2tGX7WCP2J+jMCNqw==$it/NRclMOHrfOvhAEFZ0mxIZRdbcfq
IBdwdwdDXW2dM=
```

您还需要确保隐式 iniRealm 使用一个 CredentialsMatcher 知道如何执行安全散列密码比较。所以配置的 [main] 部分:

```
[main]
...
passwordMatcher = org.apache.shiro.authc.credential.PasswordMatcher
iniRealm.credentialsMatcher = $passwordMatcher
...
```

MD5 checksum 校验和

虽然在 JVM上您可以用任意算法执行任何散列,但默认的散列算法是 MD5,常用于文件校验和。 只使用 - r (或 ——资源)选项显示下面的值是一个资源位置(而不打印出你希望散列):

> java -jar shiro-tools-hasher-X.X.X-cli.jar -r RESOURCE_PATH

默认情况下 RESOURCE_PATH 将是文件路径,但您可以指定 classpath 或URL 资源通过将 classpath: 或 url: 设置为前缀。

一些例子:

```
<command> -r fileInCurrentDirectory.txt
<command> -r ../../relativePathFile.xml
<command> -r ~/documents/myfile.pdf
<command> -r /usr/local/logs/absolutePathFile.log
<command> -r url:http://foo.com/page.html
<command> -r classpath:/WEB-INF/lib/something.jar
```

19. Terminology 术语

请花 2 分钟来阅读和理解它——这很重要。眞的。这里的术语和概念在文档的任何地方都被涉及到，它将在总体上大大简化你对Shiro 和安全的理解。

由于所使用的术语使得安全可能令人困惑。我们将通过澄清一些核心概念使生活更容易，你将会看到 Shiro API 是如何很好地反映了它们：

- **Authentication** 身份验证是验证 Subject 身份的过程——实质上是证明某些人是否眞的是他们所说的他们是谁。当认证尝试成功后，应用程序能够相信该subject 被保证是其所期望的。
- **Authorization** 授权，又称为访问控制，是决定一个 user/Subject 是否被允许做某事的过程。它通常是通过检查和解释 Subject的角色和权限（见下文），然后允许或拒绝到一个请求的资源或功能来完成的。
- **Ciphern** 密码是进行加密或解密的一种算法。该算法一般依赖于一块被称为 key 的信息。基于不同的key 的加密算法也是不一样的，所有解密没有它是非常困难的。 密码有不同的表现形式。分组密码致力于符号块，通常是固定大小的，而流密码致力于连续的符号流。对称性密码加密和解密使用相同的密钥（key），而非对称性加密使用不同的密钥。如果非对称性加密的密钥不能从其他地方得到，那么可以创建公钥/私钥对公开共享。
- **Credential** 凭证是一块信息，用来验证 user/Subject 的身份。在认证尝试期间，一个（或多个）凭证与 Principals(s)被一同提交，来验证 user/Subject 所提交的确实是所关联的用户。证书通常是非常秘密的东西，只有特定的 user/Subject 才知道，如密码或 PGP 密钥或生物属性或类似的机制。 这个想法是为 principal 设置的，只有一个人会知道正确的证书来“匹配”该 principal。如果当前 user/Subject 提供了正确的凭证匹配了存储在系统中的，那么系统可以假定并信任当前user/Subject 是眞的他们所说的他们是谁。信任度随着更安全的凭证类型加深（如，生物识别签名 > 密码）。
- **Cryptography** 加密是保护信息不受不希望的访问的习惯做法，通过隐藏信息或将它转化成无意义的东西，这样没人可以理解它。Shiro 致力于加密的两个核心要素：加密数据的密码，如使用公钥或私钥的邮件，以及散列表（也称消息摘要），它对数据进行不可逆的加密，如密码。
- **Hash** 散列函数是单向的，不可逆转的输入源，有时也被称为消息，在一个编码的哈希值内部，有时也被称为消息摘要。它通常用于密码，数字指纹，或以字节数组为基础的数据。

- **Permission** 权限，至少按照 Shiro 的解释，是在应用程序中描述原始功能的一份声明并没有更多的功能。权限是在安全策略中最低级别的概念。它们仅定义了应用程序能够做“什么”。它们没有说明“谁”能够执行这些操作。权限只是行为的声明，仅此而已。 一些权限的例子：
    - 打开文件
    - 浏览'/user/list'页面
    - 打印文档
    - 删除'jsmith'用户
- **Principal** Principal 是一个应用程序用户（Subject）的任何标志属性。“标志属性”可以是任何对你应用程序有意义的东西——用户名，姓，名，社会安全号码，用户ID 等。这就是它——没什么古怪的。Shiro 也引用一些我们称之为Subject 的 primary principal 的东西。一个 primary principal 是在整个应用程序中唯一标识 Subject 的 principal。理想的 primary principal 是用户名或 RDBMS 用户表主键——用户 ID。对于在应用 程序中的用户（Subject）来说，只有一个primary principal

- **Realm** Realm 是一个能够访问应用程序特定的安全数据（如用户，角色和权限）的组件。它可以被看作是一个特定安全的 DAO（Data Access Object）。Realm 将这些应用程序特定的数据转换成 Shiro 能够理解的格式，这样Shiro 反过来能够提供一个单一的易于理解的 Subject 编程API，无论有多少数据源存在或无论你的数据是什么样的应用程序特定的格式。 Realm 通常和数据源是一对一的对应关系，如关系数据库，LDAP 目录，文件系统，或其他类似资源。因此，Realm 接口的实现使用数据源特定的API 来展示授权数据（角色，权限等），如JDBC，文件IO，Hibernate 或 JPA，或其他数据访问API。

- **Role** 基于你对话的对象，一个角色的定义是可以多变的。在许多应用程序中，它充其量是个模糊不清的概念，人们用它来隐式定义安全策略。Shiro 偏向于把角色简单地解释为一组命名的权限的集合。这就是它——一个应用程序的唯一名称，聚集一个或多个权限声明。 这是一个比许多应用程序使用的隐式的定义更为具体的定义。如果你选择了你的数据模型反映Shiro 的假设，你会发现将有更多控制安全策略的权力。

- **Session** 会话是一个在一段时间内有状态的数据，其上下文与一个单一的与软件系统交互的user/Subject 相关联。当 Subject 使用应用程序时，能够从会话中添加/读取/删除数据，并且应用程序稍后能够在需要的地方使用该数据。会话会被终止，由于user/Subject 注销或会话不活动而超时。 对于那些熟悉 HttpSession 的，Shiro Session 服务于同一目标，除了Shiro 会话能够在任何环境下使用，甚至在没有Servlet 容器或EJB 容器的环境。

- **Subject** Subject 只是一个精挑细选的安全术语，基本上的意思是一个应用程序用户的安全特定的“视图”。然而 Subject 不总是需要反映为一个人——它可以代表一个调用你应用程序的外部进程，或许是一个系统帐户的守护进程， 在一段时间内执行一些间歇性的东西（如一个cron job）。它基本上是任何使用应用程序做某事的实体的一个代表。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

20. 10 Minute Tutorial 十分钟教程

Introduction 前言

欢迎来到 Apache Shiro 十分钟教程！

希望通过这个简单、快速的示例，可以让你对应用程序中使用 Shiro 有个深入的了解。嗯，10分钟你应该可以搞定它。

Overview 概述

Apache Shiro 是什么？

Apache Shiro 一个功能强大，使用简单的 Java 安全框架，它为开发人员提供一个直观而全面的认证，授权，加密及会话管理的解决方案。

实际上，Shiro 的主要功能是管理应用程序中与安全相关的全部，同时尽可能支持多种实现方法。Shiro 是建立在完善的接口驱动设计和面向对象原则之上的，支持各种自定义行为。Shiro 提供的默认实现，使其能完成与其他安全框架同样的功能，这不也是我们一直努力想要得到的吗！

那么 Apache Shiro 能用来做什么呢？

很多，很多，嘿嘿。但是不在快速指南中做介绍，如果你想知道，那怎么办呢？去这里找寻你的答案吧。当然如果你还想知道我们什么时候，以及为什么要“创造”Shiro，去看看Shrio的历史和使命吧。

现在让我们动手做点儿什么吧。

*Shiro*可以在任何环境下运行，小到最简单的命令行应用，大到大型的企业应用以及集群应用。但是我们准备在快速指南中使用最最简单的 *main* 方法的方式，让你对 *Shiro* 的*API*有个感官的认识。

Download 下载

1. 确保已经安装了 JDK1.5+ 和 Maven2.2+
2. 去这里下载最新已发布的源码。例子中我们使用 1.1.0 发布版本。
3. 解压源代码

    unzip shiro-root-1.2.0-source-release.zip

4. 进入快速指南文件夹

   cd shiro-root-1.1.0/samples/quickstart

5. 运行快速指南

   mvn compile exec:java

过程中会输出日志信息，用来告诉你正在进行的是什么，最后退出执行。可以在这里 samples/quickstart/src/main/java/Quickstart.java 找到源码，也可以进行修改，记得修改后运行 mvn compile exec:java 即可。

Quickstart.java

Quickstart.java 中包含刚刚我们提到的所有内容(认证、授权等等)，通过这个简单的示例可以让你轻松的熟悉Shiro的API。那么，让我们把Quickstart.java中的代码，一点一点剖析，这样便于理解它们的作用。 几乎所有的环境下，都可以通过这种方式获取当前用户：

```
Subject currentUser = SecurityUtils.getSubject();
```

通过 SecurityUtils.getSubject()，就可以获取当前 Subject。Subject 是应用中用户的一个特定安全的缩影，虽然感觉上直接使用 User 会更贴切，但是实际上它的意义远远超过了 User。而且每个应用程序都会有自己的用户以及框架，我们可不想和它们混淆在一起，况且 Subject 就是安全领域公认的名词。OK，我们继续。

在单应用系统中，调用 getSubject() 会返回一个 Subject，它是位于应用程序中特定位置的用户信息；在服务器中运行的情况下(比如web应用)，getSubject 会返回一个位于当前线程或请求中的用户信息。 现在你已经得到了 Subject 对象，那么用它可以做什么呢？

如果你想得到应用中用户当前 Session 的其他参数，可以这样获取Session 对象：

```
Session session = currentUser.getSession();
session.setAttribute( "someKey", "aValue" );
```

这个Session 对象是Shiro中特有的对象，它和我们经常使用的HttpSession 非常相似，但还提供了额外的东西，其中与 HttpSession最大的不同就是 Shiro 中的 Session 不依赖 HTTP 环境(换句话说，可以在非 HTTP 容器下运行)。

如果将 Shiro 部署在 web 应用程序中，那么这个 Session 就是基于HttpSession 的。但是像 QuickStart 示例那样，在非 web 环境下使用，Shiro 则默认使用 EnterpriseSessionManagment。也就是说，不论在应用中的任何一层使用同样的API，却不

需要考虑部署环境，这一优点为应用打开一个全新的世界，因为应用中要获取Session对象再也不用依赖于 HttpSession 或者 EJB 的会话 Bean。而且任何客户端技术都可以共享 session 数据。

现在你可以得到当前 Subject 和它的 Session 对象。那么我们如何验证比如角色和权限这些东西呢？

很简单，可以通过已得到的 user 对象进行验证。Subject 对象代表当前用户，但是，谁才是当前用户呢？他们可是匿名用户啊。也就是说，必须登录才能获取到当前用户。没问题，这样就可以搞定：

```
if ( !currentUser.isAuthenticated() ) {
    //收集用户的主要信息和凭据，来自GUI中的特定的方式
    //如包含用户名/密码的HTML表格，X509证书，OpenID，等。
    //我们将使用用户名/密码的例子因为它是最常见的。.
    UsernamePasswordToken token = new UsernamePasswordToken("lonestarr", "vespa");

    //支持'remember me' (无需配置，内建的!):
    token.setRememberMe(true);

    currentUser.login(token);
}
```

就是这样，不能再简单了。

但如果登录失败了呢，你可以捕获所有异常然后按你期望的方式去处理：

```
try {
    currentUser.login( token );
    //无异常，说明就是我们想要的!
} catch ( UnknownAccountException uae ) {
    //username 不存在，给个错误提示?
} catch ( IncorrectCredentialsException ice ) {
    //password 不匹配，再输入?
} catch ( LockedAccountException lae ) {
    //账号锁住了，不能登入。给个提示?
}
    ... 更多类型异常 ...
} catch ( AuthenticationException ae ) {
    //未考虑到的问题 - 错误?
}
```

这里有许多不同类别的异常你可以检测到，也可以抛出你自己异常。详见 AuthenticationException JavaDoc

小贴士：

最好的方式是将普通的失败信息反馈给用户，你总不会希望帮助黑客来攻击你的系统吧。

好，到现在为止，我们有了一个登录用户，接下来我们还可以做什么？

让我们显示他们是谁

```
//打印主要信息 (本例子是 username):
log.info( "User [" + currentUser.getPrincipal() + "] logged in successfully." );
```

我们也可以判断他们是否拥有某个角色：

```
if ( currentUser.hasRole( "schwartz" ) ) {
    log.info("May the Schwartz be with you!" );
} else {
    log.info( "Hello, mere mortal." );
}
```

我们也可以判断他们是否拥有某个特定动作或入口的权限：

```
if ( currentUser.isPermitted( "lightsaber:weild" ) ) {
    log.info("You may use a lightsaber ring.  Use it wisely.");
} else {
    log.info("Sorry, lightsaber rings are for schwartz masters only.");
}
```

同样，我们还可以执行非常强大的 instance-level （实例级别）的权限检测，检测用户是否具备访问某个类型特定实例的权限：

```
if ( currentUser.isPermitted( "winnebago:drive:eagle5" ) ) {
    log.info("You are permitted to 'drive' the 'winnebago' with license plate (id) 'ea
gle5'.  " +
                "Here are the keys - have fun!");
} else {
    log.info("Sorry, you aren't allowed to drive the 'eagle5' winnebago!");
}
```

轻而易举，是吧！

最后，当用记不再使用系统，可以退出登录：

```
currentUser.logout(); //清楚识别信息，设置 session 失效.
```

这些就是使用 Apache Shiro 开发应用的核心了，当然，Apache Shiro已将将很多复杂的东西封装在内部了，但是现在它就是这么简单。

你会有疑问吧，用户登录时，谁负责把用户信息(用户名、密码、角色、权限等)取出来，还有运行时，谁负责安全认证呢？当然由你决定了啊。通过将一个实现了 Shiro 中的 Realm 的 Reaml 配置到 Shiro 中即可。

至于如何配置很大程度上取决于你的运行时环境，比如在单应用、web 应用、基于 Spring 或 JEE 容器的应用或者组合模式中使用 Shiro，配置都有所不同。如何配置已经超出 QuickStart 示例的范围，因为它的主要目的是帮助你熟悉 Shiro 的 API 和概念。

如果想进一步了解 Shiro，可以看看 Authentication Guide 和 Authorization Guide。也可以查看其他文档(特别是 用户指南)，这里可以解决你的各种疑问。

感谢一路同行，希望你能喜欢使用 Apache Shiro。

译者注：本文参考：http://shiro.apache.org/10-minute-tutorial.html。如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

21. Beginner's Webapp Tutorial 初学者web应用教程

本文是一篇循序渐进介绍用 Apache Shiro 保护 web 应用程序的教程。 它假定读者已经具备了 Shiro 的入门知识,并假设至少熟悉以下两个介绍性文档:

- 用Shiro保护你的应用安全
- Apache Shiro 十分钟教程

学习本教程应该需要45分钟到1个小时时间。 当你完成后,你将有一个很好的关于 Shiro 是如何在一个 web 应用程序的概念。

目录

Overview 概述

虽然 Apache Shiro 的核心设计目标允许它被用于任何基于 java 的应用程序的安全,如命令行应用程序、服务器守护进程 ,web 应用程序,等等,本指南将专注于最常见的用例:确保 web 应用程序安全运行在一个 servlet 容器,例如 Tomcat 或 Jetty。

Prerequisites 先决条件

以下工具将被安装在本地开发机器为了跟随本教程。

- Git(测试版 w/1.7)
- Java SDK 7
- Maven 3
- 你最喜欢的IDE,比如 IntelliJ IDEA 或 Eclipse ,甚至一个简单的文本编辑器用于查看文件和更改。

Tutorial Format 教程格式

这是一个循序渐进的教程。 本教程,和它的所有步骤,存在Git存储库。 当你复制 git 存储库, master 分支是你的起点。 在教程的每一步都是一个独立的分支。 你可以跟随只需查看 git 分支反映本教程一步你审查

The Application 应用程序

我们将构建的 web 应用程序是一个超级网络应用,可以作为一个起点为您自己的应用程序。它将展示用户登录,注销,特定于用户的欢迎消息,访问控制web 应用程序的某些部分,pluggable 安全数据存储和集成。

我们将开始通过建立项目,包括构建工具和声明依赖性,以及配置 servlet的 web.xml 文件启动 web 应用程序和 Shiro 的环境。

一旦我们完成设置,我们将层的各个部分的功能,包括集成的安全数据存储,然后让用户登录,注销,访问控制。

Project Setup项目设置

不必手动设置一个目录结构和初始基本文件,我们已为你这样做好了一个 git 存储库。

1. Fork the tutorial project 先fork本教程项目

在 github，浏览 tutorial project 项目,点击 Fork 按钮

2. Clone your tutorial repository 复制教程存储库

现在您已经将项目 fork 在你的 GitHub 帐户,克隆它在本地机器上:

> $ git clone git@github.com:$YOUR_GITHUB_USERNAME/apache-shiro-tutorial-webapp.git

(其中 $YOUR_GITHUB_USERNAME 是你的 GitHub 用户名)

用 cd 进入本地的项目目录查看项目结构:

> $ cd apache-shiro-tutorial-webapp

3. Review project structure 审查项目结构

当前项目结构为：

```
apache-shiro-tutorial-webapp/
  |-- src/
  |   |-- main/
  |     |-- resources/
  |       |-- logback.xml
  |     |-- webapp/
  |       |-- WEB-INF/
  |         |-- web.xml
  |       |-- home.jsp
  |       |-- include.jsp
  |       |-- index.jsp
  |-- .gitignore
  |-- .travis.yml
  |-- LICENSE
  |-- README.md
  |-- pom.xml
```

解释下：

- pom.xml :Maven 项目/构建文件。 它有Jetty 配置,这样你就可以马上运行 mvn jetty:run 测试您的 web 应用程序运行。
- README.md :一个简单的项目的自述文件
- LICENSE :该项目是 Apache 2.0 许可协议
- .travis.yml :一个 Travis CI 配置文件以确保它总是在项目构建时，持续运行集成构建您的项目。
- .gitignore :一个 git 忽略文件,包含的后缀和目录是那些不应该纳入到版本控制中。
- src/main/resources/logback.xml:一个简单的 Logback 配置文件。 对于本教程,我们选择 SLF4J 的日志 API 和 Logback 日志的实现。 这可能很容易被熟悉 Log4J 或者 JUL 的人所接受。
- src/main/webapp/WEB-INF/web.xml :最初的 web.xml 文件,我们将配置很快使用Shiro。
- src/main/webapp/include.jsp :一个页面,其中包含常见的引入和声明,包括其他的JSP页面。 这让我们在一个地方来管理引入和声明。
- src/main/webapp/home.jsp :应用的简单的默认主页。 包括 include.jsp (如将其他人,因为我们很快就会看到)。
- src/main/webapp/index.jsp :默认站点索引页面-这仅仅是将请求转发给我们 home.jsp 主页。

4. Run the webapp 运行

运行

> $ mvn jetty:run

打开浏览器访问 localhost:8080,页面将会输出 Hello, World!

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 1: Enable Shiro 启动 shiro

我们最初的 master 库 只是一个简单的通用的 web 应用程序,可以作为任何应用程序的模板。让我们添加的最低限度,启动 Shiro web 应用程序。

执行以下git checkout 命令加载 Step1 分支:

> $ git checkout step1

检出的分支，有两点变化

1. 添加了一个 src/main/webapp/WEB-INF/shiro.ini 文件
2. src/main/webapp/WEB-INF/web.xml 改变了.

1a: Add a shiro.ini file

可以配置 Shiro 在许多不同的方式在一个web应用程序,这取决于您所使用的web和/或MVC框架。 例如,您可以通过Spring配置Shiro,Guice,Tapestry,和许多更多。

为了简单起见,我们将启动一个 Shiro 环境使用Shiro的默认值(非常简单的) INI 配置 。

如果你签出 Step1 分支,您将看到这个新的的内容 src/main/webapp/WEB-INF/shiro.ini 文件(简短标题删除注释):

```
[main]

# Let's use some in-memory caching to reduce the number of runtime lookups against Sto
rmpath.
# A real application might want to use a more robust caching solution (e.g. ehcache or
 a
# distributed cache).  When using such caches, be aware of your cache TTL settings: to
o high
# a TTL and the cache won't reflect any potential changes in Stormpath fast enough.  T
oo low
# and the cache could evict too often, reducing performance.
cacheManager = org.apache.shiro.cache.MemoryConstrainedCacheManager
securityManager.cacheManager = $cacheManager
```

ini 包含一个简单的 [main] 和一些最小的配置:

- 它定义了一个新的 cacheManager (缓存管理器) 实例。 缓存是Shiro的体系结构的一个重要组成部分,它减少了不断往返通信各种数据存储。 这个示例使用 MemoryConstrainedCacheManager 这是唯一眞正好的单个JVM 的应用程序。 如果您的应用程序部署在多个主机(如集群网络服务器),您需要使用集群缓存管理器实现。

- 在Shiro securityManager 它配置新 cacheManager (缓存管理器) 的实例 。 一个Shiro SecurityManager 实例总是存在的,所以它不需要显式地定义。

1b: Enable Shiro in web.xml

当我们有一个 shiro.ini 配置,我们需要加载它,并开始一个新的 Shiro 环境和使 web 应用程序环境的实现。

我们所做的这一切通过添加现有的几件事到 src/main/webapp/WEB-INF/web.xml 文件:

```
<listener>
    <listener-class>org.apache.shiro.web.env.EnvironmentLoaderListener</listener-class
>
</listener>

<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>org.apache.shiro.web.servlet.ShiroFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

1c: Run the webapp

当检出 step1 分支，运行

> $ mvn jetty:run

这一次,你会看到日志输出类似于以下,表明 Shiro 确实是运行在你的应用:

```
16:04:19.807 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Starting Shiro environ
ment initialization.
16:04:19.904 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Shiro environment init
ialized in 95 ms.
```

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 2: Connect to a User Store 连接用户存储

检出 step2 分支

> $ git checkout step2

现在我们已经在 webapp 中集成和运行了 Shiro。 但是我们还没有真正告诉 Shiro 做任何事!

之前我们可以登录,注销,或执行基于角色或基于许可的访问控制,或任何其他安全相关的,我们需要用户!

我们需要配置 Shiro 访问 用户存储 的一些类型的,所以它可以查找用户执行登录尝试,或检查角色的安全决策,等等。有许多类型的用户存储任何应用程序可能需要访问:也许你在 MySQL 数据库中存储用户,也许在MongoDB,也许你的公司将用户帐户存储在 LDAP 或 Active Directory,也许你将它们存储在一个简单的文件,或其他专有数据存储。

Shiro 通过所谓的 Realm 来实现这些。 Shiro 的文档:

> Reamls 是 Shiro 和你的程序安全数据之间的“桥”或者“连接”，它用来实际和安全相关的数据如用户执行身份认证（登录）的帐号和授权（访问控制）进行交互，Shiro 从一个或多个程序配置的 Realm 中查找这些东西。 Realm 本质上是一个特定的安全 DAO：它封装与数据源连接的细节，得到Shiro 所需的相关的数据。在配置 Shiro 的时候，你必须指定至少一个Realm 来实现认证（authentication）和/或授权（authorization）。SecurityManager 可以配置多个复杂的 Realm，但是至少有一个是需要的。 Shiro 提供开箱即用的 Realms 来连接安全数据源（或叫地址）如 LDAP、JDBC、文件配置如INI和属性文件等，如果已有的Realm不能满足你的需求你也可以开发自己的Realm实现。 和其它内部组件一样，Shiro SecurityManager 管理如何使用 Realms获取 Subject 实例所代表的安全和身份信息。(译者注：详见说明文档)

因此,我们需要配置一个领域,那么我们可以访问用户。

2a: Set up Stormpath

本教程的精神是保持尽可能简单,不引入复杂性或范围干扰了我们的学习Shiro 的目的,我们将使用一个简单的 realm : Stormpath realm。

Stormpath 云托管用户管理服务,以完全自由发展为目的。 这意味着启用 Stormpath 之后,你已经准备好如下:

- 一个用户界面来管理应用程序,目录,帐户和组。 Shiro 不提供这个,所以通过本教程这将是方便和节省你的时间。
- 一个安全的存储用户密码的机制。 您的应用程序不需要担心密码安全、密码比较或存储密码。 虽然 Shiro 可以做这些事情,你必须配置它们,知道密码的概念。 Stormpath 自动化密码安全所以你(Shiro)不需要担心如何“步入正轨”。
- 过电子邮件帐户 电子邮件验证和密码重置的安全工作流通。 Shiro不支持这个,因为它通常是特定于应用程序的。
- 主持/管理”always on“基础设施——你不需要设置任何或维持任何东西。

对于本教程,Stormpath 比建立一个独立的 RDBMS 服务器还有担心 SQL 或密码加密问题等等简单的多了。 所以我们将使用它。

当然,Stormpath 只是许多 Shiro 可以连接的后端数据存储之一。 我们将讨论更复杂的数据存储和特定于应用程序的配置之后。

Sign up for Stormpath 注册

1. 填写 Stormpath 注册表单, 它会发邮件确认
2. 确认邮件

Get a Stormpath API Key 获取API Key

Stormpath API 所需的关键是 Stormpath Realm 用来与 Stormpath 交流。 获得 Stormpath API Key:

1. 登录到 Stormpath 管理控制台 使用你的Stormpath 注册使用的电子邮件地址和密码
2. 在结果页面的右上角,访问 Settings > My Account 。
3. 在账户信息页面, Security Credentials, 点击 Create API Key 。

这将生成 API Key 并下载到你的电脑 apiKey.properties 文件。 如果你在文本编辑器中打开文件,您将看到类似于下面的:

```
apiKey.id = 144JVZINOF5EBNCMG9EXAMPLE
apiKey.secret = lWxOiKqKPNwJmSldbiSkEbkNjgh2uRSNAb+AEXAMPLE
```

将该文件保存在一个安全的位置,比如在一个隐藏您的主目录 .stormpath 目录中。 例如:

```
$HOME/.stormpath/apiKey.properties
```

还改变文件权限,以确保只有你能读这个文件。 例如,在 *nix 操作系统:

```
$ chmod go-rwx $HOME/.stormpath/apiKey.properties
```

Register the web application with Stormpath 注册web应用

我们必须通过 Stormpath 注册我们的 web 应用程序，用于用户的管理和身份验证。简单通过 REST请求, POST 到一个新的 Stormpath 应用程序资源 URL:

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    -d '{
           "name" : "Apache Shiro Tutorial Webapp"
        }' \
    'https://api.stormpath.com/v1/applications?createDirectory=true'
```

其中

- $YOUR_API_KEY_ID 是 apiKey.properties 文件中 apiKey.id 值
- YOUR_API_KEY_SECRET 是 apiKey.properties 文件中 apiKey.secret 值

那样就将会创建你的应用，下面是一个响应示例：

```
{
    "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe",
    "name": "Apache Shiro Tutorial Webapp",
    "description": null,
    "status": "ENABLED",
    "tenant": {
        "href": "https://api.stormpath.com/v1/tenants/sOmELoNgRaNDoMIdHeRe"
    },
    "accounts": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/accoun
ts"
    },
    "groups": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/groups
"
    },
    "loginAttempts": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeR/loginAt
tempts"
    },
    "passwordResetTokens": {
        "href": "https://api.stormpath.com/v1/applications/aLoNGrAnDoMAppIdHeRe/passwo
rdResetTokens"
    }
}
```

注意顶层的 href ,如 https://api.stormpath.com/v1/applications/$YOUR_APPLICATION_ID ，接下来我们将在 shiro.ini 配置使用这个 href 。

Create an application test user account 创建应用测试用户账号

现在有了应用，我们要创建一个简单的测试用户

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    -d '{
           "givenName": "Jean-Luc",
           "surname": "Picard",
           "username": "jlpicard",
           "email": "capt@enterprise.com",
           "password":"Changeme1"
        }' \
 "https://api.stormpath.com/v1/applications/$YOUR_APPLICATION_ID/accounts"
```

同样的，不要忘了修改 $YOUR_APPLICATION_ID

2b: Configure the Realm in shiro.ini 配置

一旦你选择至少一个用户连接存储,我们将需要配置一个 Realm 来表示数据存储,然后告诉 Shiro SecurityManager 。

如果你已经签出了 step2 分支,你会注意到的 shiro.ini 文件的 (主要) 现在部分有以下补充:

```
# 配置 Realm 来连接用户存储.本教程只简单的指向 Stormpath
# 花 5 分钟进行设置:
stormpathClient = com.stormpath.shiro.client.ClientFactory
stormpathClient.cacheManager = $cacheManager
stormpathClient.apiKeyFileLocation = $HOME/.stormpath/apiKey.properties
stormpathRealm = com.stormpath.shiro.realm.ApplicationRealm
stormpathRealm.client = $stormpathClient

# 找到这个 你在Stormpath 创建应用时的 URL :
# Applications -> (choose application name) --> Details --> REST URL
stormpathRealm.applicationRestUrl = https://api.stormpath.com/v1/applications/$STORMPA
TH_APPLICATION_ID
stormpathRealm.groupRoleResolver.modeNames = name
securityManager.realm = $stormpathRealm
```

做以下修改:

- 改变 $ HOME 占位符实际主目录路径,例如 /home/jsmith 所以最后 stormpathClient.apiKeyFileLocation 值是类似 /home/jsmith/.stormpath/apiKey.properties 。 这条路必须匹配的位置 apiKey.properties 你在 Step 2a.中从Stormpath下载一个文件。
- 改变 step2 中 Stormpath 返回来的 href 中 $STORMPATH_APPLICATION_ID 占位符中的实际ID值。 最后的 stormpathRealm.applicationRestUrl 值应该类似 https://api.stormpath.com/v1/applications/6hsPwoRZ0hCk6ToytVxi4D (当然有不同的应

用程序ID)。

2c: Commit your changes 提交修改

替换 $ HOME 和 STORMPATH_APPLICATION_ID 值是特定于您的应用程序。 继续提交这些更改你的分支:

```
$ git add . && git commit -m "updated app-specific placeholders" .
```

2d: Run the webapp

运行

```
$ mvn jetty:run
```

看到如下输出：

```
16:08:25.466 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Starting Shiro environ
ment initialization.
16:08:26.201 [main] INFO  o.a.s.c.IniSecurityManagerFactory - Realms have been explici
tly set on the SecurityManager instance - auto-setting of realms will not occur.
16:08:26.201 [main] INFO  o.a.shiro.web.env.EnvironmentLoader - Shiro environment init
ialized in 731 ms.
```

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 3: Enable Login and Logout 启用登录、登出

现在我们有了用户，可以简单的再 UI 里面增加、删除、禁用他们。现在我们要用到登录、登出功能了。

检出 step3 分支

```
$ git checkout step3
```

这次检出的内容，增加了下面两项：

- 新增了一个登录界面 src/main/webapp/login.jsp 包含一个简单的登录框，让我们登入
- shiro.ini 文件更新了，从而能支持 web (URL) 特性

Step 3a: Enable Shiro form login and logout support

step3 分支中 src/main/webapp/WEB-INF/shiro.ini 文件包含了下面两个内容:

```
[main]

shiro.loginUrl = /login.jsp

# Stuff we've configured here previously is omitted for brevity

[urls]
/login.jsp = authc
/logout = logout
```

shiro.* lines

其中 shiro.loginUrl = /login.jsp 这个是设置 Shiro 的登录页面是 /login.jsp

设置 Shiro 的默认 authc filter (默认是 FormAuthenticationFilter) 识别这个登录页面. 这使得 FormAuthenticationFilter 能够正常工作

The [urls] section

[urls] 是一个新的 web 特性的 INI

这部分允许您使用一个非常简洁的名称/值对语法告诉 shiro 如何过滤请求任何给定的 UR L路径。 所有的路径 [url] 相对于web应用程序的[HttpServletRequest.getContextPath())(http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/HttpServletRequest.html getContextPath()))的值。

这些名称/值对提供了一个非常强大的方式来过滤请求,允许各种各样的安全规则。 更深入的报道 url 和过滤器链超出了本文的范围,但请做 阅读更多关于它, 如果你感兴趣。

现在,我们将讨论的两行补充道:

/login.jsp = authc /logout = logout

- 第一行意思"每当 Shiro 看到 /login.jsp 的 url 请求,都将会在请求中启用 Shiro authc 过滤器"。
- 第二行意味着"每当 Shiro 看到 /logout 的 url 请求,都将会在请求中启用 Shiro 注销过滤器。"

这两个过滤器是有点特别的:他们实际上并不需要背后的东西。 而不是过滤,他们会完全处理请求。 这就意味着什么都不用为这些 url 请求 做什么——不用写控制器! Shiro 将处理这些请求。

Step 3b: Add a login page

从 step3 启用登录和注销的支持,现在我们需要确保实际上有一个 /login.jsp 页面显示一个登录表单。

step3 分支包含一个新 src/main/webapp/login.jsp 页面。 这是一个简单的足够 bootstrap 风格的 HTML 登录页面,但有四个重要的事情:

- form 的 action 值是空字符串。当一个 form 不包含 action 值，则浏览器将会提交 form 请求到相同的 URL。这是可以的,因为我们将告诉 Shiro 该 URL 是什么，所以 Shiro 可以快速自动处理任何登录提交。 /login.jsp = authc 这句是告诉我们 authc 过滤器去处理提交。
- 有一个 username 表单字段。 Shiro authc 过滤器自动寻找 username 的请求参数 在 login 提交时，并且在登录期间使用那个值 (很多 Realms 允许 这个可以是 email 或者是 username).
- 有一个 password 表单字段。 Shiro authc 过滤器自动寻找 password 的请求参数 在 login 提交时。
- 有一个 rememberMe 的 checkbox，当选中时，值表示 '是' (如 true, t, 1, enabled, y, yes 等等).

我们的 login.jsp 表单只使用默认值 username , password , rememberme 表单字段的名称。名称是可配置的,如果你希望改变他们，看 FormAuthenticationFilter 的 javadoc 获取信息。

Step 3c: Run the webapp

```
$ mvn jetty:run
```

Step 3d: Try to Login

浏览器 访问 localhost:8080/login.jsp ,就能看到登录界面

输入 Step 2 中的 登录名称、密码。点击登录，成功则去到主页，失败则返回到登录界面

如果想登录后去到跟主页不同的任意界面，可以设置 authc.successUrl = /任意页面 ，即可

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 4: User-specific UI changes

如果想实现界面——当前登录用户是谁——这个功能的话，只需要简单用到 shiro 的 jsp 标签

检出 step4 分支

$ git checkout step4

在 home.jsp 中就几个新增内容:

- 当用户没有登录，则在登录页面显示 ‘Welcome Guest’.
- 当用户登入，则能看到用户名称 ‘Welcome username’ 并且有一个登出的链接t.
- 这个 UI 是非常常见的，操作按钮在屏幕右上方.

Step 4a: Add the Shiro Tag Library Declaration

home.jsp 修改包含下面内容:

```
<%@ taglib prefix="shiro" uri="http://shiro.apache.org/tags" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

允许使用 Core (c:) 和 Shiro (shiro:) 两个 jsp 标签库

Step 4b: Add Shiro Guest and User tags

修改 home.jsp 包含下面 和 两个标签

```
p>Hi <shiro:guest>Guest</shiro:guest><shiro:user>
<%
    //This should never be done in a normal page and should exist in a proper MVC cont
roller of some sort, but for this
    //tutorial, we'll just pull out Stormpath Account data from Shiro's PrincipalColle
ction to reference in the
    //<c:out/> tag next:

    request.setAttribute("account", org.apache.shiro.SecurityUtils.getSubject().getPri
ncipals().oneByType(java.util.Map.class));

%>
<c:out value="${account.givenName}"/></shiro:user>!
    ( <shiro:user><a href="<c:url value="/logout"/>">Log out</a></shiro:user>
    <shiro:guest><a href="<c:url value="/login.jsp"/>">Log in</a></shiro:guest> )
</p>
```

看起来有点难懂：

- : 这个标签只显示 Shiro Subject 在应用里面是 ‘guest’时的内容. Shiro 定义了 guest 是任何 没有登录系统，或者没有被前次登录记住的 Subject (使用 Shiro ‘remember me’ 功能).
- : 这个标签只显示 Shiro Subject 在应用里面是 ‘user’时的内容. Shiro 定义了 user 是任何登录系统，或者被前次登录记住的 Subject (使用 Shiro ‘remember me’ 功能).

上面的代码片段将呈现以下,如果 Subject 是 guest:

Hi Guest! (Log in)

其中“Log in”是一个超链接到 /login.jsp

它将呈现以下，如果 Subject 是 user:

Hi jsmith! (Log out)

假设 jsmith 的帐户的用户名登录。 “Log out”是一个超链接到 ‘/logout’ 注销 过滤器。

正如您可以看到的,你可以关掉或整个页面上部分,特性和 UI 组件。 除了 和 , Shiro 支持许多其他有用的 JSP 标签 ,您可以使用自定义 UI 知道当前基于各种各样的 Subject。

Step 4c: Run the webapp

运行:

```
$ mvn jetty:run
```

访问 localhost:8080用 guest 身份, 而后登录。成功登录后，看页面显示，知道你先是一个用户了！

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 5: Allow Access to Only Authenticated Users

虽然您可以更改页面内容基于 Subject 的状态,很多时候你会想要限制你的整个部分应用基于是否有人 证明 他们的身份(身份验证)在他们当前的与 web 应用程序的交互。

这是特别重要的,如果一个用户只能部分的应用显示敏感信息,如帐单细节或控制其他用户的能力。

执行以下git checkout 命令 加载 step5 分支:

```
$ git checkout step5
```

Step 5 包含下面三点变化:

- 我们添加了新的部分 (url 路径) ，想限制只有通过身份验证的用户。
- 改变了 shiro.ini 告诉 shiro 只允许经过身份验证的用户应用 web 应用程序的一部分
- 改变了主页，输出内容是基于 当前 Subject 是否被验证.

Step 5a: Add a new restricted section

新的 src/main/webapp/account 目录添加进来了。这个目录及下面的目录，只有登录用户可见。 src/main/webapp/account/index.jsp 文件只是一个占位符一个模拟 “home account” 页面。

Step 5b: Configure shiro.ini

shiro.ini 修改了：

```
/account/** = authc
```

这 过滤器链定义的意思是“任何请求访问 /account (或其子路径) 必须经过身份验证”。

但是如果有人试图访问路径或它的任何孩子路径?

但是你还记得在 step3 中添加以下行来 (主要) 部分:

shiro.loginUrl = /login.jsp

这条自动配置 authc 与我们的 webapp 的登录 URL 过滤器。

基于这一配置, authc 过滤器已经足够聪明知道如果当前 Subject 访问 /account 时还没有经过身份验证 ,它将自动重定向到 /login.jsp 页面。 成功登录后,它会自动将用户重定向回他们试图访问的页面(/account)。 方便!

Step 5c: Update our home page

修改 /home.jsp 页面让用户知道他们可以访问新网站的一部分。添加欢迎以下信息:

```
<shiro:authenticated><p>Visit your <a href="<c:url value="/account"/>">account page</a
>.</p></shiro:authenticated>
<shiro:notAuthenticated><p>If you want to access the authenticated-only <a href="<c:ur
l value="/account"/>">account page</a>,
    you will need to log-in first.</p></shiro:notAuthenticated>
```

标签只会显示内容如果当前 Subject 已经登录在当前会话(身份验证)。 这是为什么 Subject 知道他们可以访问新网站的一部分。

标签只会显示内容如果当前 Subject 在当前还没有经过身份验证的会话。

但你注意到 notAuthenticated 还有一个URL的内容 /account ? 没关系,我们 authc 过滤器将处理 登录并且重定向 的流程。如上所述。

试一试!

Step 5d: Run the webapp

```
$ mvn jetty:run
```

访问 localhost:8080, 点击 /account 链接重定向强制你登录。成功登录后，看页面显示，知道你已经登录了！尝试登录、登出。

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 6: Role-Based Access Control 基于角色的访问控制

除了控制访问身份验证的基础上,它通常是一个要求限制访问应用程序的某些部分基于角色分配给当前 Subject

检出 step6

```
$ git checkout step6
```

Step 6a: Add Roles

为了实现基于角色的访问控制,我们需要角色存在。

本教程是最快的方式来填充一些内容到 Stormpath。

要做到这一点,登录到用户界面和导航如下:

Directories > Apache Shiro Tutorial Webapp Directory > Groups

添加下面组

- Captains
- Officers
- Enlisted

(这个是 Star-Trek《星际迷航》 里面的角色)

一旦你创建了组,添加 Jean-Luc Picard 到 Captains 和 Officers 分组。 您可能想要创建一些特别账户,并将它们添加到您喜欢的任何组。 确保一些帐户不重叠组,这样你就可以看到变化基于单独的组分配到用户帐户。

Step 6b: RBAC Tags

修改 home.jsp 内容

```
<h2>Roles</h2> section of the home page:

<h2>Roles</h2>

<p>Here are the roles you have and don't have. Log out and log back in under different
 user
    accounts to see different roles.</p>

<h3>Roles you have:</h3>

<p>
    <shiro:hasRole name="Captains">Captains<br/></shiro:hasRole>
    <shiro:hasRole name="Officers">Bad Guys<br/></shiro:hasRole>
    <shiro:hasRole name="Enlisted">Enlisted<br/></shiro:hasRole>
</p>

<h3>Roles you DON'T have:</h3>

<p>
    <shiro:lacksRole name="Captains">Captains<br/></shiro:lacksRole>
    <shiro:lacksRole name="Officers">Officers<br/></shiro:lacksRole>
    <shiro:lacksRole name="Enlisted">Enlisted<br/></shiro:lacksRole>
</p>
```

标签只会显示内容如果当前 Subject 分配指定的角色。

如果当前标签只会显示内容 Subject 没有 被分配指定的角色。

Step 6c: RBAC filter chains

留给读者的练习(不是定义步骤)是创建一个新的部分的网站和限制的URL访问部分网站基于角色分配给当前用户 。

Step 6d: Run the webapp

```
$ mvn jetty:run
```

访问 localhost:8080, 和不同的用户帐户登录看主页的分配不同的角色， 角色部分内容改变!

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Step 7: Permission-Based Access Control 基于权限的访问控制

基于角色的访问控制是好的,但它存在一个主要问题:你不能在运行时添加或删除角色。 角色名字在角色检查是硬编码的,所以如果你改变了角色名称或角色配置,或添加或删除角色,你必须回去和改变你的代码!

正因为如此,Shiro 有强大的特点:内置的支持 权限 。 Shiro 的 权限是一个原始语句的功能,例如 'open a door' 'create a blog entry', 'delete the jsmith user'等权限反映了应用的原始功能,所以当你改变应用的功能时你只需要更改权限的检查——而不是改变你的角色或用户模型。

为了证明这一点,我们将创建一些权限,并将它们分配给一个用户,然后定制我们的基于用户的授权(权限) web UI

Step 7a: Add Permissions

Shiro Realms 是只读的组件:每个数据存储模型的角色,组织、权限、账号,以及它们之间的关系不同,所以 Shiro 没有"写"API来修改这些资源。 修改底层模型对象,你只是通过任何 API 直接修改你想要的。

是这样的,因为我们使用 Stormpath 在这个示例应用程序中,我们将权限分配给一个帐户和组在 Stormpath API-specific 方式。

让我们执行 cURL 请求添加一些权限给我们以前创建的 Jean-Luc Picard 帐户。 使用该帐户的 href URL,我们会请求 apacheShiroPermissions 账户通过 自定义数据 :

```
curl -X POST --user $YOUR_API_KEY_ID:$YOUR_API_KEY_SECRET \
    -H "Accept: application/json" \
    -H "Content-Type: application/json" \
    -d '{
            "apacheShiroPermissions": [
                "ship:NCC-1701-D:command",
                "user:jlpicard:edit"
            ]
        }' \
"https://api.stormpath.com/v1/accounts/$JLPICARD_ACCOUNT_ID/customData"
```

$JLPICARD_ACCOUNT_ID 匹配 创建Jean-Luc Picard 时 的 uid

添加两个权限到 Stormpath 账户:

- ship:NCC-1701-D:command
- user:jlpicard:edit

第一句含义是，你有权限 'command' 编号是 'NCC-1701-D' 的 'ship'。 第二句是有权限 edit 账户是 jlpicard 的 user

想知道 Stormpath 是如何存储权限的，请参阅 Shiro Stormpath plugin documentation.

Step 7b: Permission Tags

就像我们对角色检查 JSP 标记,并行标记存在权限检查。 我们更新 /home.jsp 页面,让用户知道如果他们允许做一些基于权限分配给他们。 这些消息被添加在一个新的

Permissions

部分的主页:

```
    <h2>Permissions</h2>
    <ul>
        <li>You may <shiro:lacksPermission name="ship:NCC-1701-D:command"><b>NOT</b> </
shiro:lacksPermission> command the <code>NCC-1701-D</code> Starship!</li>
        <li>You may <shiro:lacksPermission name="user:${account.username}:edit"><b>NOT
</b> </shiro:lacksPermission> edit the ${account.username} user!</li>
    </ul>
```

当你访问主页，看到如下输出

You may NOT command the NCC-1701-D Starship! You may NOT edit the user!

当用 Jean-Luc Picard 账户登录 ,您将看到这个:

You may command the NCC-1701-D Starship! You may edit the user!

使用 shiro WildcardPermission 中的语法

您可以看到,Shiro解决,通过身份验证的用户权限,和输出在一个适当的方式呈现。

你也可以使用 标记为肯定的权限检查。

最后,我们将呼吁关注一个非常强大的功能和权限检查。 你看到第二个使用权限检查，如何运行时生成权限值?

...

${account.username} 在运行时解释,形成最终的值 user:aUsername:edit,然后最后一个字符串值是用于权限检查。

这是极强大的:你可以执行权限检查基于当前用户是谁, 目前正在与什么交互。 这些基于运行时 runtime实例级权限检查基本技术发展高度可定制的、安全的应用程序。

Step 7c: Run the webapp

```
$ mvn jetty:run
```

访问 localhost:8080 并用 Jean-Luc Picard 账户(和其他账户)登录、登出,看到基于权限分配的页面输出变化(或者木有)

按 `ctl-C` (或者 mac 中的 `cmd-C`) 来关闭应用

Summary 总结

我们希望你发现这个入门教程 Shiro-enabled webapps有用。 在未来版本的教程中,我们将介绍:

插入不同的用户数据存储,如 RDBMS 或 NoSQL 数据存储。

Fixes and Pull Requests 修复和 pull 请求

请发送任何修复勘误表作为 GitHub pull 请求 到 https://github.com/lhazlewood/apache-shiro-tutorial-webapp 存储库。 我们很感激!!!

译者注:本文参考:http://shiro.apache.org/webapp-tutorial.html。如果对本中文翻译有疑议的或发现勘误欢迎指正,点此提问。

22. Application Security With Apache Shiro 用Shiro保护你的应用安全

在尝试保护你的应用时，你是否有过挫折？是否觉得现有的 Java 安全解决方案难以使用，只会让你更糊涂？本文介绍的 Apache Shiro，是一个不同寻常的 Java 安全框架，为保护应用提供了简单而强大的方法。本文还解释了 Apache Shiro 的项目目标、架构理念以及如何使用 Shiro 为应用安全保驾护航。

What is Apache Shiro?？

Apache Shiro（发音为“shee-roh”，日语“堡垒（Castle）”的意思）是一个强大易用的 Java 安全框架，提供了认证、授权、加密和会话管理功能，可为任何应用提供安全保障——从命令行应用、移动应用到大型网络及企业应用。

Shiro 为解决下列问题提供了保护应用的API（我喜欢称它们为应用安全的四大基石）：

- **Authentication**（认证）：用户身份识别，通常被称为用户“登录”
- **Authorization**（授权）：访问控制。
- **Cryptography**（加密）：保护或隐藏数据防止被偷窥。
- **Session Management**（会话管理）：每用户相关的时间敏感的状态。

Shiro 还支持一些辅助特性，如 Web 应用安全、单元测试和多线程，它们的存在强化了上面提到的四个要素。

Why was Apache Shiro created 为何要创建 Shiro？

对于一个框架来讲，使其有存在价值的最好例证就是有让你去用它的原因，它应该能完成一些别人无法做到的事情。要理解这一点，需要了解 Shiro 的历史以及创建它时的其他替代方法。

在 2008 年加入 Apache 软件基金会之前，Shiro 已经5岁了，之前它被称为 JSecurity 项目，始于2003年初。当时，对于 Java 应用开发人员而言，没有太多的通用安全替代方案 - 我们被 Java 认证/授权服务（或称为 JAAS ）紧紧套牢了。JAAS 有太多的缺点 - 尽管它的认证功能尚可忍受，但授权方面却显得拙劣，用起来令人沮丧。此外，JAAS 跟虚拟机层面的安全问题关系非常紧密，如判断 JVM 中是否允许装入一个类。作为应用开发者，我更关心应用最终用户能做什么，而不是我的代码在 JVM 中能做什么。

由于当时正从事应用开发，我也需要一个干净、容器无关的会话机制。在当时，“这场游戏”中唯一可用的会话是 HttpSessions，它需要 Web 容器；或是 EJB 2.1 里的有状态会话 Bean，这又要 EJB 容器。而我想要的一个与容器脱钩、可用于任何我选择的环境中的会话。

最后就是加密问题。有时，我们需要保证数据安全，但是 Java 密码架构（Java Cryptography Architecture）让人难以理解，除非你是密码学专家。API 里到处都是 Checked Exception，用起来很麻烦。我需要一个干净、开箱即用的解决方案，可以在需要时方便地对数据加密/解密。

于是，纵观 2003 年初的安全状况，你会很快意识到还没有一个大一统的框架满足所有上述需求。有鉴于此，JSecurity（即之后的 Apache Shiro）诞生了。

Why would you use Apache Shiro today 今天，你为何愿意使用 Shiro？

从 2003 年至今，框架选择方面的情况已经改变了不少，但今天仍有令人信服的理由让你选择 Shiro。其实理由相当多，Apache Shiro：

- 易于使用 - 易用性是这个项目的最终目标。应用安全有可能会非常让人糊涂，令人沮丧，并被认为是“necessary evil(必要之恶)”。若是能让它简化到新手都能很快上手，那它将不再是一种痛苦了。
- 广泛性 - 没有其他安全框架可以达到 Apache Shiro 宣称的广度，它可以为你的安全需求提供“一站式”服务。
- 灵活性 - Apache Shiro 可以工作在任何应用环境中。虽然它工作在Web、EJB 和 IoC 环境中，但它并不依赖这些环境。Shiro 既不强加任何规范，也无需过多依赖。
- **Web**能力 - Apache Shiro 对 Web 应用的支持很神奇，允许你基于应用 URL 和 Web 协议（如 REST ）创建灵活的安全策略，同时还提供了一套控制页面输出的 JSP 标签库。
- 可插拔 - Shiro 干净的 API 和设计模式使它可以方便地与许多的其他框架和应用进行集成。你将看到 Shiro 可以与诸如 Spring、Grails、Wicket、Tapestry、Mule、Apache Camel、Vaadin这类第三方框架无缝集成。
- 支持 - Apache Shiro 是 Apache 软件基金会成员，这是一个公认为了社区利益最大化而行动的组织。项目开发和用户组都有随时愿意提供帮助的友善成员。像 Katasoft 这类商业公司，还可以给你提供需要的专业支持和服务。

Who’s Using Shiro 谁在用Shiro?

Shiro 及其前身 JSecurity 已被各种规模和不同行业的公司项目采用多年。自从成为 Apache 软件基金会的顶级项目后，站点流量和使用呈持续增长态势。许多开源社区也正在用 Shiro，这里有些例子如 Spring，Grails，Wicket，Tapestry，Tynamo，Mule和Vaadin。

如Katasoft，Sonatype，MuleSoft这样的商业公司，一家大型社交网络和多家纽约商业银行都在使用 Shiro 来保护他们的商业软件和站点。

Core Concepts: Subject, SecurityManager, and Realms 核心概念

已经描述了Shiro的好处，那就让我们看看它的API，好让你能够有个感性认识。Shiro 架构有三个主要概念 - Subject，SecurityManager和Realms。

Subject

在考虑应用安全时，你最常问的问题可能是“当前用户是谁？”或“当前用户允许做什么吗？”。当我们写代码或设计用户界面时，问自己这些问题很平常：应用通常都是基于用户故事构建的，并且你希望功能描述（和安全）是基于每个用户的。所以，对于我们而言，考虑应用安全的最自然方式就是基于当前用户。Shiro 的 API 用它的 Subject 概念从根本上体现了这种思考方式。

Subject 一词是一个安全术语，其基本意思是“当前的操作用户”。称之为“用户”并不准确，因为“用户”一词通常跟人相关。在安全领域，术语“Subject”可以是人，也可以是第三方进程、后台帐户（Daemon Account）或其他类似事物。它仅仅意味着“当前跟软件交互的东西”。但考虑到大多数目的和用途，你可以把它认为是 Shiro 的“用户”概念。在代码的任何地方，你都能轻易的获得 Shiro Subject，参见如下代码：

Listing 1. Acquiring the Subject 获得Subject

```
import org.apache.shiro.subject.Subject;
import org.apache.shiro.SecurityUtils;
...
Subject currentUser = SecurityUtils.getSubject();
```

一旦获 得Subject，你就可以立即获得你希望用 Shiro为 当前用户做的90% 的事情，如登录、登出、访问会话、执行授权检查等——稍后还会看到更多。这里的关键点是Shiro的API非常直观，因为它反映了开发者以‘每个用户’思考安全控制的自然趋势。同时，在代码的任何地方都能很轻松地访问Subject，允许在任何需要的地方进行安全操作。

SecurityManager

Subject 的“幕后”推手是 SecurityManager。Subject 代表了当前用户的安全操作，SecurityManager 则管理所有用户的安全操作。它是 Shiro框架的核心，充当“保护伞”，引用了多个内部嵌套安全组件，它们形成了对象图。但是，一旦 SecurityManager 及其内部对象

图配置好，它就会退居幕后，应用开发人员几乎把他们的所有时间都花在 Subject API调用上。

那么，如何设置 SecurityManager 呢？嗯，这要看应用的环境。例如，Web 应用通常会在 Web.xml 中指定一个 Shiro Servlet Filter，这会创建 SecurityManager 实例，如果你运行的是一个独立应用，你需要用其他配置方式，但有很多配置选项。

一个应用几乎总是只有一个 SecurityManager 实例。它实际是应用的Singleton（尽管不必是一个静态 Singleton）。跟 Shiro 里的几乎所有组件一样，SecurityManager 的缺省实现是 POJO，而且可用 POJO 兼容的任何配置机制进行配置 - 普通的Java代码、Spring XML、YAML、.properties 和 .ini 文件等。基本来讲，能够实例化类和调用JavaBean 兼容方法的任何配置形式都可使用。

为此，Shiro 借助基于文本的INI配置提供了一个缺省的“公共”解决方案。INI 易于阅读、使用简单并且需要极少依赖。你还能看到，只要简单地理解对象导航，INI 可被有效地用于配置像 SecurityManager 那样简单的对象图。注意，Shiro 还支持 Spring XML 配置及其他方式，但这里只我们只讨论INI。

下列清单2列出了基于INI的Shiro最简配置：

Listing 2. Configuring Shiro with INI 用INI配置Shiro

```
[main]
cm = org.apache.shiro.authc.credential.HashedCredentialsMatcher
cm.hashAlgorithm = SHA-512
cm.hashIterations = 1024
# Base64 encoding (less text):
cm.storedCredentialsHexEncoded = false
iniRealm.credentialsMatcher = $cm

[users]
jdoe = TWFuIGlzIGRpc3Rpbmd1aXNoZWQsIG5vdCBvbmx5IGJpcyByZWFzb2
asmith = IHNpbmd1bGFyIHBhc3Npb24gZnJvbSBvdGhlciBhbXNoZWQsIG5vdCB
```

在清单2中，我们看到了用于配置 SecurityManager 实例的 INI 配置例子。有两个INI段落：[main]和[users].

[main]段落是配置 SecurityManager 对象及其使用的其他任何对象（如Realms）的地方。在示例中，我们看到配置了两个对象：

1. cm对象，是 Shiro 的 HashedCredentialsMatcher 类实例。如你所见，cm 实例的各属性是通过“嵌套点”语法进行配置的 - 在清单3中可以看到I niSecurityManagerFactory 使用的惯例，这种方法代表了对象图导航和属性设置。
2. iniRealm 对象，它被 SecurityManager 用来表示以 INI 格式定义的用户帐户。

[users]段落是指定用户帐户静态列表的地方 - 为简单应用或测试提供了方便。

就介绍而言，详细了解每个段落的细节并不是重点。相反，看到 INI 配置是一种配置 Shiro 的简单方式才是关键。关于 INI 配置的更多细节，请参见 Shiro 文档。

Listing 3. Loading shiro.ini Configuration File 加载ini配置文件

```
import org.apache.shiro.SecurityUtils;
import org.apache.shiro.config.IniSecurityManagerFactory;
import org.apache.shiro.mgt.SecurityManager;
import org.apache.shiro.util.Factory;

...

//1.加载INI配置
Factory<SecurityManager> factory = new IniSecurityManagerFactory("classpath:shiro.ini"
);

//2. 创建SecurityManager
SecurityManager securityManager = factory.getInstance();

//3. 使其可访问
SecurityUtils.setSecurityManager(securityManager);
```

在清单3的示例中，我们看到有三步：

1. 加装用来配置 SecurityManager 及其构成组件的 INI 配置文件；
2. 根据配置创建 SecurityManager 实例（使用 Shiro 的工厂概念，它表述了工厂模式）；
3. 使应用可访问 SecurityManager Singleton。在这个简单示例中，我们将它设置为 VM 静态 Singleton，但这通常不是必须的——你的应用配置机制可以决定你是否需要使用静态存储。

Realms

Realm 充当了 Shiro 与应用安全数据间的“桥梁”或者“连接器”。也就是说，当切实与像用户帐户这类安全相关数据进行交互，执行认证（登录）和授权（访问控制）时，Shiro 会从应用配置的 Realm 中查找很多内容。

从这个意义上讲，Realm 实质上是一个安全相关的 DAO：它封装了数据源的连接细节，并在需要时将相关数据提供给Shiro。当配置 Shiro 时，你必须至少指定一个Realm，用于认证和（或）授权。配置多个 Realm 是可以的，但是至少需要一个。

Shiro 内置了可以连接大量安全数据源（又名目录）的 Realm，如LDAP、关系数据库（JDBC）、类似INI的文本配置资源以及属性文件等。如果缺省的Realm不能满足需求，你还可以插入代表自定义数据源的自己的Realm实现。下面的清单4是通过INI配置Shiro使用LDAP目录作为应用Realm的示例。

Listing 4. Example realm configuration snippet to connect to LDAP user data store Realm配置示例片段：连接存储用户数据的LDAP

```
[main]
ldapRealm = org.apache.shiro.realm.ldap.JndiLdapRealm
ldapRealm.userDnTemplate = uid={0},ou=users,dc=mycompany,dc=com
ldapRealm.contextFactory.url = ldap://ldapHost:389
ldapRealm.contextFactory.authenticationMechanism = DIGEST-MD5
```

既然已经了解如何建立一个基本的Shiro环境，下面让我们来讨论，作为一名开发者该如何使用这个框架。

Authentication认证

认证是核实用户身份的过程。也就是说，当用户使用应用进行认证时，他们就在证明他们就是自己所说的那个人。有时这也理解为“登录”。它是一个典型的三步骤过程。

1. 收集用户的身份信息，称为用户的身份（principal），以及身份的支持证明，称为证明（Credential）。
2. 将用户的身份和证书提交给系统。
3. 如果提交的证书与系统期望的该用户身份（当事人）匹配，该用户就被认为是经过认证的，反之则被认为未经认证的。

这个过程的常见例子是大家都熟悉的“用户/密码”组合。多数用户在登录软件系统时，通常提供自己的用户名（用户的身份）和支持他们的密码（证书）。如果存储在系统中的密码（或密码表示）与用户提供的匹配，他们就被认为通过认证。

Shiro 以简单直观的方式支持同样的流程。正如我们前面所说，Shiro 有一个以 Subject 为中心的API - 几乎你想要用 Shiro 在运行时完成的所有事情都能通过与当前执行的Subject进行交互而达成。因此，要登录Subject，只需要简单地调用它的login方法，传入表示被提交当事人和证书（在这种情况下，就是用户名和密码）的 AuthenticationToken实例。示例如清单5中所示：

Listing 5. Subject Login Subject登录

```
//1. 接受提交的用户身份和证明：
AuthenticationToken token =
new UsernamePasswordToken(username, password);

//2. 获取当前Subject：
Subject currentUser = SecurityUtils.getSubject();
//3. 登录：
currentUser.login(token);
```

你可以看到，Shiro 的 API 很容易地就反映了这个常见流程。你将会在所有的 Subject 操作中继续看到这种简单风格。在调用了 login 方法后，SecurityManager 会收到 AuthenticationToken，并将其发送给已配置的Realm，执行必须的认证检查。每个 Realm 都能在必要时对提交的AuthenticationTokens 作出反应。但是如果登录失败了会发生什么？如果用户提供了错误密码又会发生什么？通过对 Shiro 的运行时AuthenticationException 做出反应，你可以控制失败，参见清单6。

Listing 6. Handle Failed Login 处理失败的登录

```
//3. 登录：
try {
    currentUser.login(token);
} catch (IncorrectCredentialsException ice) {
    …
} catch (LockedAccountException lae) {
    …
}
…
catch (AuthenticationException ae) {…
}
```

你可以选择捕获 AuthenticationException 的一个子类，作出特定的响应，或者对任何 AuthenticationException 做一般性处理（例如，显示给用户普通的“错误的用户名或密码”这类消息）。选择权在你，可以根据应用需要做出选择。

Subject 登录成功后，他们就被认为是已认证的，通常你会允许他们使用你的应用。但是仅仅证明了一个用户的身份并不意味着他们可以对你的应用为所欲为。这就引出了另一个问题，“我如何控制用户能做或不能做哪些事情？”，决定用户允许做哪些事情的过程被称为授权。下面我们将谈谈 Shiro如何进行授权。

Authorization授权

授权实质上就是访问控制——控制用户能够访问应用中的哪些内容，比如资源、Web 页面等等。多数用户执行访问控制是通过使用诸如角色和权限这类概念完成的。也就是说，通常用户允许或不允许做的事情是根据分配给他们的角色或权限决定的。那么，通过检查这些角色和权限，你的应用程序就可以控制哪些功能是可以暴露的。如你期望的，Subject API 让你可以很容易的执行角色和权限检查。如清单7中的代码片段所示：如何检查 Subject被分配了某个角色：

Listing 7. Role Check 角色检查

```
if ( subject.hasRole(“administrator”) ) {
    //显示‘Create User’按钮
} else {
    //按钮置灰?
}
```

如你所见，你的应用程序可基于访问控制检查打开或关闭某些功能。

权限检查是执行授权的另一种方法。上例中的角色检查有个很大的缺陷：你无法在运行时增删角色。角色名字在这里是硬编码，所以，如果你修改了角色名字或配置，你的代码就会乱套！如果你需要在运行时改变角色含义，或想要增删角色，你必须另辟蹊径。

为此，Shiro 支持了权限（permissions）概念。权限是功能的原始表述，如‘开门’，‘创建一个博文’，‘删除‘jsmith’用户’等。通过让权限反映应用的原始功能，在改变应用功能时，你只需要改变权限检查。进而，你可以在运行时按需将权限分配给角色或用户。

如清单8中，我们重写了之前的用户检查，取而代之使用权限检查。

Listing 8. Permission Check 权限检查

```
if ( subject.isPermitted(“user:create”) ) {
    //显示‘Create User’按钮
} else {
    //按钮变成灰?
}
```

这样，任何具有“user:create”权限的角色或用户都可以点击‘Create User’按钮，并且这些角色和指派甚至可以在运行时改变，这给你提供了一个非常灵活的安全模型。

“user:create”字符串是一个权限字符串的例子，它遵循特定的解析惯例。Shiro 借助它的 WildcardPermission 支持这种开箱即用的惯例。尽管这超出了本文的范围，你会看到在创建安全策略时，WildcardPermission非常灵活，甚至支持像实例级别访问控制这样的功能。

清单9. 实例级别的权限检查

```
if ( subject.isPermitted(“user:delete:jsmith”) ) {
    //删除‘jsmith’用户
} else {
    //不删除‘jsmith’
}
```

该例表明，你可以对你需要的单个资源进行访问控制，甚至深入到非常细粒度的实例级别。如果愿意，你甚至还可以发明自己的权限语法。参见 Shiro Permission 文档可以了解更多内容。最后，就像使用认证那样，上述调用最终会转向 SecurityManager，它会咨询Realm做出自己的访问控制决定。必要时，还允许单个Realm同时响应认证和授权操作。

以上就是对Shiro授权功能的简要概述。虽然多数安全框架止于授权和认证，但 Shiro 提供了更多功能。下面，我们将谈谈 Shiro 的高级会话管理功能。

Session Management会话管理

在安全框架领域，Apache Shiro 提供了一些独特的东西：可在任何应用或架构层一致地使用 Session API。即，Shiro 为任何应用提供了一个会话编程范式——从小型后台独立应用到大型集群 Web 应用。这意味着，那些希望使用会话的应用开发者，不必被迫使用 Servlet 或 EJB 容器了。或者，如果正在使用这些容器，开发者现在也可以选择使用在任何层统一一致的会话API，取代 Servlet 或 EJB 机制。

但 Shiro 会话最重要的一个好处或许就是它们是独立于容器的。这具有微妙但非常强大的影响。例如，让我们考虑一下会话集群。对集群会话来讲，支持容错和故障转移有多少种容器特定的方式？Tomcat 的方式与 Jetty 的不同，而 Jetty 又和 Websphere 不一样，等等。但通过 Shiro 会话，你可以获得一个容器无关的集群解决方案。Shiro 的架构允许可插拔的会话数据存储，如企业缓存、关系数据库、NoSQL 系统等。这意味着，只要配置会话集群一次，它就会以相同的方式工作，跟部署环境无关 - Tomcat、Jetty、JEE 服务器或者独立应用。不管如何部署应用，毋须重新配置应用。

Shiro 会话的另一好处就是，如果需要，会话数据可以跨客户端技术进行共享。例如，Swing 桌面客户端在需要时可以参与相同的Web应用会话中 - 如果最终用户同时使用这两种应用，这样的功能会很有用。那你如何在任何环境中访问Subject的会话呢？请看下面的示例，里面使用了Subject的两个方法。

Listing 10. Subject’s Session 会话

```
Session session = subject.getSession();
Session session = subject.getSession(boolean create);
```

如你所见，这些方法在概念上等同于 HttpServletRequest API。第一个方法会返回 Subject 的现有会话，或者如果还没有会话，它会创建一个新的并将之返回。第二个方法接受一个布尔参数，这个参数用于判定会话不存在时是否创建新会话。一旦获得 Shiro 的会话，你几乎可以像使用HttpSession 一样使用它。Shiro 团队觉得对于 Java 开发者，HttpSession API用起来太舒服了，所以我们保留了它的很多感觉。当然，最大的不同在于，你可以在任何应用中使用 Shiro 会话，不仅限于Web 应用。清单11中显示了这种相似性。

Listing 11. Session methods 会话的方法

```
Session session = subject.getSession();
session.getAttribute("key", someValue);
Date start = session.getStartTimestamp();
Date timestamp = session.getLastAccessTime();
session.setTimeout(millis); ...
```

Cryptography 加密

加密是隐藏或混淆数据以避免被偷窥的过程。在加密方面，Shiro的目标是简化并让JDK的加密支持可用。

清楚一点很重要，一般情况下，加密不是特定于Subject的，所以它是Shiro API的一部分，但并不特定于Subject。你可以在任何地方使用Shiro的加密支持，甚至在不使用Subject的情况下。对于加密支持，Shiro真正关注的两个领域是加密哈希（又名消息摘要）和加密密码。下面我们来看看这两个方面的详细描述。

Hashing哈希

如果你曾使用过 JDK 的 MessageDigest 类，你会立刻意识到它的使用有点麻烦。MessageDigest 类有一个笨拙的基于工厂的静态方法 API，它不是面向对象的，并且你被迫去捕获那些永远都不必捕获的 Checked Exceptions。如果需要输出十六进制编码或 Base64 编码的消息摘要，你只有靠自己 - 对上述两种编码，没有标准的 JDK 支持它们。Shiro 用一种干净而直观的哈希 API 解决了上述问题。

打个比方，考虑比较常见的情况，使用 MD5 哈希一个文件，并确定该哈希的十六进制值。被称为‘校验和’，这在提供文件下载时常用到 - 用户可以对下载文件执行自己的MD5哈希。如果它们匹配，用户完全可以认定文件在传输过程中没有被篡改。

不使用 Shiro，你需要如下步骤才能完成上述内容：

1. 将文件转换成字节数组。JDK 中没有干这事的，故而你需要创建一个辅助方法用于打开 FileInputStream，使用字节缓存区，并抛出相关的IOExceptions，等等。
2. 使用 MessageDigest 类对字节数组进行哈希，处理相关异常，如清单12所示。
3. 将哈希后的字节数组编码成十六进制字符。JDK 中还是没有干这事的，你依旧需要创建另外一个辅助方法，有可能在你的实现中会使用位操作和位移动。

Listing 12. JDK's MessageDigest JDK的消息摘要

```
try {
    MessageDigest md = MessageDigest.getInstance("MD5");
    md.digest(bytes);
    byte[] hashed = md.digest();
} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

对于这样简单普遍的需求，这个工作量实在太大了。现在看看Shiro是如何做同样事情的：

```
String hex = new Md5Hash(myFile).toHex();
```

当使用 Shiro 简化所有这些工作时，一切都非常简单明了。完成SHA-512哈希和密码的Base64编码也一样简单。

```
String encodedPassword = new Sha512Hash(password, salt, count).toBase64();
```

你可以看到 Shiro 对哈希和编码简化了不少，挽救了你处理在这类问题上所消耗的脑细胞。

Ciphers密码

加密是使用密钥对数据进行可逆转换的加密算法。我们使用其保证数据的安全，尤其是传输或存储数据时，以及在数据容易被窥探的时候。

如果你曾经用过 JDK 的 Cryptography API，特别是javax.crypto.Cipher 类，你会知道它是一头需要驯服的极其复杂的野兽。对于初学者，每个可能的加密配置总是由一个javax.crypto.Cipher实例表示。必须进行公钥/私钥加密？你得用 Cipher。需要为流操作使用块加密器（Block Cipher）？你得用 Cipher。需要创建一个AES 256位Cipher 来保护数据？你得用 Cipher。你懂的。

那么如何创建你需要的 Cipher 实例？您得创建一个非直观、标记分隔的加密选项字符串，它被称为“转换字符串（transformation string）”，把该字符串传给 Cipher.getInstance 静态工厂方法。这种字符串方式的cipher 选项，并没有类型安全以确保你正在用有效的选项。这也暗示没有JavaDoc帮你了解相关选项。并且，如果字符串格式组织不正确，你还需要进一步处理Checked Exception，即便你知道配置是正确的。如你所见，使用 JDK Cipher 是一项相当繁重的任务。很久以前，这些技术曾经是Java API 的标准，但是世事变迁，我们需要一种更简单的方法。

Shiro 通过引入它的 CipherService API 试图简化加密密码的整个概念。CipherService 是多数开发者在保护数据时梦寐以求的东西：简单、无状态、线程安全的 API，能够在一次方法调用中对整个数据进行加密或解密。你所需要做的只是提供你的密钥，就可根据需要加密或解密。如下列清单13中，使用 256 位 AES 加密：

Listing 13. Apache Shiro’s Encryption API 加密API

```
AesCipherService cipherService = new AesCipherService();
cipherService.setKeySize(256);

//创建一个测试密钥：
byte[] testKey = cipherService.generateNewKey();
//加密文件的字节：
byte[] encrypted = cipherService.encrypt(fileBytes, testKey);
```

较之JDK的Cipher API，Shiro的示例要简单的多：

- 你可以直接实例化一个 CipherService——没有奇怪或让人混乱的工厂方法；

- Cipher配置选项可以表示成 JavaBean ——兼容的 getter 和 setter 方法 - 没有了奇怪和难以理解的“转换字符串”；
- 加密和解密在单个方法调用中完成；
- 没有强加的 Checked Exception。如果愿意，可以捕获 Shiro 的CryptoException。

Shiro 的 CipherService API还有其他好处，如同时支持基于字节数组的加密/解密（称为“块”操作）和基于流的加密/解密（如加密音频或视频）。

不必再忍受 Java Cryptography 带来的痛苦。Shiro 的 Cryptography 支持就是为了减少你在确保数据安全上付出的努力。

Web Support Web支持

最后，但并非不重要，我们将简单介绍一下 Shiro 的 Web 支持。Shiro附带了一个帮助保护 Web 应用的强建的 Web 支持模块。对于 Web 应用，安装 Shiro 很简单。唯一需要做的就是在 web.xml 中定义一个 Shiro Servlet 过滤器。清单14中，列出了代码。

Listing 14. ShiroFilter in web.xml

```
<filter>
    <filter-name>ShiroFilter</filter-name>
    <filter-class>
        org.apache.shiro.web.servlet.IniShiroFilter
    </filter-class>
    <!-- 没有init-param属性就表示从classpath:shiro.ini装入INI配置 -->
</filter>
<filter-mapping>
    <filter-name>ShiroFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

这个过滤器可以读取上述 shiro.ini 配置，这样不论什么开发环境，你都拥有了一致的配置体验。一旦完成配置，Shiro Filter就会过滤每个请求并且确保在请求期间特定请求的 Subject 是可访问的。同时由于它过滤了每个请求，你可以执行安全特定的逻辑以保证只有满足一定标准的请求才被允许通过。

URL-Specific Filter Chains URL特定的Filter链

Shiro 通过其创新的URL过滤器链功能支持安全特定的过滤规则。它允许你为任何匹配的URL模式指定非正式的过滤器链。这意味着， 使用 Shiro 的过滤器机制，你可以很灵活的强制安全规则（或者规则的组合） - 其程度远远超过你单独在web.xml中定义过滤器时所获得的。清单15中显示了Shiro INI中的配置片段。

Listing 15. Path-specific Filter Chains 路径特定的Filter链

```
[urls]
/assets/** = anon
/user/signup = anon
/user/** = user
/rpc/rest/** = perms[rpc:invoke], authc
/** = authc
```

如你所见，Web 应用可以使用[urls] INI段落。对于每一行，等号左边的值表示相对上下文的Web 应用路径。等号右边的值定义了过滤器链 - 一个逗号分隔的有序 Servlet 过滤器列表，它会针对给出的路径进行执行。每个过滤器都是普通的 Servlet 过滤器，你看到的上面的过滤器名字（anon，user，perms，authc）是 Shiro 内置的安全相关的特殊过滤器。你可以搭配这些安全过滤器来创建高度定制的安全体验。你还可以指定任何其他现有的 Servlet 过滤器。

相比起使用 web.xml，在其中先定义过滤器块，然后定义单独分离的过滤器模式块，这种方式带来的好处有多少？采用Shiro的方法，可以很容易就准确知道针对给定匹配路径执行的过滤器链。如果想这么做，你可以在web.xml 中仅定义 Shiro Filter，在 shiro.ini中定义所有其他的过滤器和过滤器链，这要比 web.xml简洁得多，而且更容易理解过滤器链定义机制。即使不使用 Shiro 的任何安全特性，单凭这样小小的方便之处，也值得让你使用 Shiro。

JSP Tag Library JSP标签库

Shiro 还提供了 JSP 标签库，允许你根据当前 Subject 的状态控制 JSP 页面的输出。一个有用的常见示例是在用户登录后显示“Hello "文本。但若是匿名用户，你可能想要显示其他内容，如换而显示“Hello! Register Today!”。清单16显示了如何使用Shiro的JSP标签实现这个示例：

Listing 16. JSP Taglib Example JSP标签库示例

```
<%@ taglib prefix="shiro"
   uri="http://shiro.apache.org/tags" %>
...
<p>Hello
<shiro:user>
    <!-- shiro:principal打印出了Subject的主当事人 - 在这个示例中，就是用户名： -->
    <shiro:principal/>!
</shiro:user>
<shiro:guest>
    <!-- 没有登录 - 就认为是Guest。显示注册链接： -->
    ! <a href=”register.jsp”>Register today!</a>
</shiro:guest>
</p>
```

除了上面例子用到的标签，还有其他标签可以让你根据用户属于（或不属于）的角色，分配（或未分配）的权限，是否已认证，是否来自“记住我”服务的记忆，或是匿名访客，包含输出。

Shiro 还支持其他许多 Web 特性，如简单的“记住我”服务，REST和BASIC认证。当然，如果想使用Shiro原生的企业会话，它还提供透明的HttpSession 支持。参见 Apache Shiro Web文档可以了解更多内容。

Web Session Management Web会话管理

最后值得一提的是 Shiro 在 Web 环境中对会话的支持。

Default Http Sessions 缺省Http会话

对于Web应用，Shiro缺省将使用我们习以为常的 Servlet 容器会话作为其会话基础设施。即，当你调用 subject.getSession()和subject.getSession(boolean) 方法时，Shiro 会返回 Servlet 容器的 HttpSession 实例支持的 Session 实例。这种方式的曼妙之处在于调用 subject.getSession() 的业务层代码会跟一个 Shiro Session 实例交互 - 还没有“认识”到它正跟一个基于 Web 的HttpSession 打交道。这在维护架构层之间的清晰隔离时，是一件非常好的事情。

Shiro’s Native Sessions in the Web Tier Web层中Shiro的原生会话

如果你由于需要 Shiro 的企业级会话特性（如容器无关的集群）而打开了Shiro 的原生会话管理，你当然希望 HttpServletRequest.getSession() 和 HttpSession API 能和“原生”会话协作，而非 Servlet 容器会话。如果你不得不重构所有使用HttpServletRequest 和 HttpSession API 的代码，使用 Shiro 的Session API 来替换，这将非常令人沮丧。Shiro 当然从来不会期望你这么做。相反，Shiro 完整实现了 Servlet 规范中的 Session 部分以在Web应用中支持原生会话。这意味着，不管何时你使用相应的HttpServletRequest 或 HttpSession 方法调用，Shiro 都会将这些调用委托给内部的原生会话 API。结果，你无需修改 Web 代码，即便是你正在使用 Shiro 的‘原生’企业会话管理 - 确实是一个非常方便（且必要）的特性。

Additional Features 附加特性

Apache Shiro 框架还包含有对保护Java应用非常有用的其他特性，如：

- 为维持跨线程的Suject提供了线程和并发支持（支持Executor和ExecutorService）；
- 为了将执行逻辑作为一种特殊的Subject，支持Callable和Runnable接口；
- 为了表现为另一个Subject的身份，支持“Run As”（比如，在管理应用中有用）；
- 支持测试工具，这样可以很容易的对Shiro的安全代码进行单元测试和集成测试。

框架局限

常识告诉我们，Apache Shiro 不是“银弹” - 它不能毫不费力的解决所有安全问题。如下是 Shiro 还未解决，但是值得知道的：

- 虚拟机级别的问题：Apache Shiro 当前还未处理虚拟机级别的安全，比如基于访问控制策略，阻止类加载器中装入某个类。然而，Shiro集成现有的JVM安全操作并非白日做梦 - 只是没人给项目贡献这方面的工作。
- 多阶段认证：目前，Shiro 不支持“多阶段”认证，即用户可能通过一种机制登录，当被要求再次登录时，使用另一种机制登录。这在基于Shiro 的应用中已经实现，但是通过应用预先收集所有必需信息再跟 Shiro交互。这个功能在 Shiro 的未来版本中非常有可能得到支持。
- **Realm**写操作：目前所有 Realm 实现都支持“读”操作来获取验证和授权数据以执行登录和访问控制。诸如创建用户帐户、组和角色或与用户相关的角色组和权限这类“写”操作还不支持。这是因为支持这些操作的应用数据模型变化太大，很难为所有 的 Shiro 用户强制定义“写”API。

Upcoming Features未来的特性

Apache Shiro社 区每天都在壮大，借此，Shiro 的特性亦是如此。在即将发布的版本中，你可能会看到：

- 更干净的Web过滤机制，无需子类化就可支持更多的插件式过滤器。
- 更多可插拔的缺省Realm实现，优先采用组合而非继承。你可以插入查找认证和授权数据的组件，无需实现为 Shiro Realm的 子类；
- 强健的 OpenID 和 OAuth（可能是混合）客户端支持；
- 支持Captcha；
- 针对纯无状态应用的配置简化（如，许多REST环境）；
- 通过请求/响应协议进行多阶段认证；
- 通过AuthorizationRequest进行粗粒度的授权；
- 针对安全断言查询的ANTLR语法（比如，(‘role(admin) && (guest || !group(developer))’）

Summary 总结

Apache Shiro 是一个功能齐全、健壮、通用的Java安全框架，你可以用其为你的应用护航。通过简化应用安全的四个领域，即认证、授权、会话管理和加密，在真实应用中，应用安全能更容易被理解和实现。Shiro 的简单架构和兼容 JavaBean 使其几乎能够在任何环境下配置和使用。附加的 Web支持和辅助功能，比如多线程和测试支持，让这个框架为应用安全提供了“一站式”服务。Apache Shiro 开发团队将继续前进，精炼代码库和支持社区。随着持续被开源和商业应用采纳，可以预期 Shiro 会继续发展壮大。

资源

- Apache Shiro的主页。
- Shiro的下载页面，包含面向 Maven 和 Ant+Ivy 用户的额外信息。
- Apache Shiro的文档页面，包含指南和参考手册。
- Apache Shiro 演讲视频和幻灯，项目的PMC主席Les Hazlewood提供。
- 其他关于Apache Shiro的文章和PPT Apache Shiro的邮件列表和论坛。
- Katasoft - 提供 Apache Shiro 专业支持和应用安全产品的公司。
- 译者注：Apache Shiro 1.2.x 用户指南 ：开源项目，一个官方文档和其他文章的翻译集合

译者注：本文参考：http://www.infoq.com/articles/apache-shiro。如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

23. CacheManager 缓存管理

Shiro 有三个重要的缓存接口：

- CacheManager - 负责所有缓存的主要管理组件，它返回 Cache 实例。
- Cache - 维护key/value 对。
- CacheManagerAware - 通过想要接收和使用 CacheManager 实例的组件来实现。

CacheManager 返回Cache 实例，各种不同的 Shiro 组件使用这些Cache 实例来缓存必要的数据。任何实现了 CacheManagerAware 的 Shiro 组件将会自动地接收一个配置好的 CacheManager，该 CacheManager 能够用来获取 Cache 实例。

Shiro 的 SecurityManager 实现及所有 AuthorizingRealm 实现都实现了 CacheManagerAware 。如果你在 SecurityManager 上设置了 CacheManger，它反过来也会将它设置到实现了CacheManagerAware 的各种不同的 Realm 上（OO delegation）。例如，在 shiro.ini 中：

```
securityManager.realms = $myRealm1, $myRealm2, ..., $myRealmN
...
cacheManager = my.implementation.of.CacheManager
...
securityManager.cacheManager = $cacheManager
# at this point, the securityManager and all CacheManagerAware
# realms have been set with the cacheManager instance
```

我们有开箱即用的 EhCacheManager实现 ,所以马上就能使用它。否则，您也可以很好的实现自己的 CacheManager （例如 Coherence，等），配置如上。

Authorization Cache Invalidation 授权缓存失效

最后请注意, AuthorizingRealm 有一个 clearCachedAuthorizationInfo) 方法能够被子类调用，用来清除特殊账户缓存的授权信息。它通常被自定义逻辑调用，如果与之匹配的账户授权数据发生了改变（来确保下次的授权检查能够捕获新数据）。

为文档加把手

我们希望这篇文档可以帮助你使用 Apache Shiro 进行工作，社区一直在不断地完善和扩展文档，如果你希望帮助 Shiro 项目，请在你认为需要的地方考虑更正、扩展或添加文档，你提供的任何点滴帮助都将扩充社区并且提升 Shiro。

提供你的文档的最简单的途径是将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。

24. Apache Shiro Cryptography Features 加密功能

Cryptography is the practice of protecting information from undesired access by hiding it or converting it into nonsense so know one else can read it. Shiro focuses on two core elements of Cryptography: ciphers that encrypt data like email using a public or private key, and hashes (aka message digests) that irreversibly encrypt data like passwords.

Shiro Cryptography's primary goal is take what has traditionally be an extremely complex field and make it easy for the rest of us while providing a robust set of cryptography features.

Simplicity Features

Interface-driven, POJO based - All of Shiro's APIs are interface-based and implemented as POJOs. This allows you to easily configure Shiro Cryptography components with JavaBeans-compatible formats like JSON, YAML, Spring XML and others. You can also override or customize Shiro as you see necessary, leveraging its API to save you time and effort. Simplified wrapper over JCE - The Java Cryptography Extension (JCE) can be complicated and difficult to use unless you're a cryptography expert. Shiro's Cryptography APIs are much easier to understand and use, and they dramatically simplify JCE concepts. So now even Cryptography novices can find what they need in minutes rather than hours or days. And you won't sacrifice any functionality because you still have access to more complicated JCE options if you need them. “Object Orientifies” cryptography concepts - The JDK/JCE's Cipher and Message Digest (Hash) classes are abstract classes and quite confusing, requiring you to use obtuse factory methods with type-unsafe string arguments to acquire instances you want to use. Shiro 'Object Orientifies' Ciphers and Hashes, basing them on a clean object hierarchy, and allows you to use them by simple instantiation. Runtime Exceptions - Like everywhere else in Shiro, all cryptography exceptions are RuntimeExceptions. You can decide whether or not to catch an exception based on your needs. Cipher Features

OO Hierarchy - Unlike the JCE, Shiro Cipher representations follow an Object-Oriented class hierarchy that match their mathematical concepts: AbstractSymmetricCipherService, DefaultBlockCipherService, AesCipherService, etc. This allows you to easily override existing classes and extend functionality as needed. Just instantiate a class - Unlike the JCE's confusing factory methods using String token arguments, using Shiro Ciphers are much easier - just instantiate a class, configure it with JavaBeans properties as necessary, and use it as desired. For example, new AesCipherService(). More secure default settings - The JCE Cipher instances assume a 'lowest common denominator' default and do not

automatically enable more secure options. Shiro will automatically enable the more secure options to ensure your data is as safe as it can be by default, helping you prevent accidental security holes. Hash Features

Deault interface implementations - Shiro provides default Hash (aka Message Digests in the JDK) implementations out-of-the-box, such as MD5, SHA1, SHA-256, et al. This provides a type-safe construction method (e.g. new Md5Hash(data)) instead of being forced to use type-unsafe string factory methods in the JDK. Built-in Hex and Base64 conversion - Shiro Hash instances can automatically provide Hex and Base-64 encoding of hashed data via their toHex() and toBase64() methods. So now you do not need to figure out how to correctly encode the data yourself. Built-in Salt and repeated hashing support - Salts and repeated hash iterations are very valuable tools when hashing data, especially when it comes to protecting user passwords. Shiro's Hash implementations support salts and multiple hash iterations out of the box so you don't have to repeat this logic anywhere you might need it.

Shiro 十分钟入门教程

跟着10 Minute Tutorial 十分钟教程进行 Shiro 的尝试。如果有任何问题将它发送到用户论坛或邮件列表

译者注：如果对本中文翻译有疑议的或发现勘误欢迎指正，点此提问。